Vol. 4 No. 8

acm queue

architecting tomorrow's computing

October 2006

Major Release Fatigue

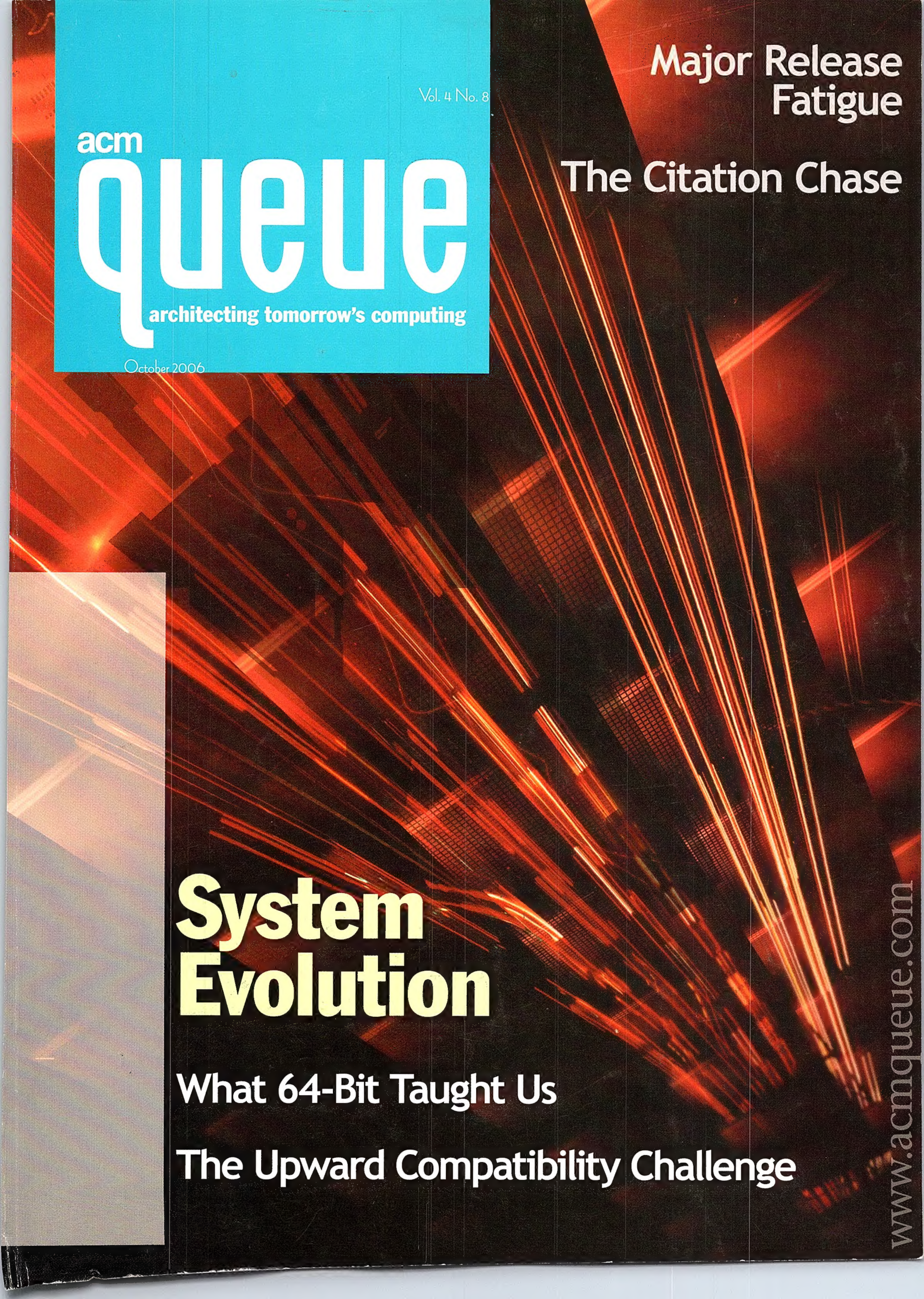
The Citation Chase

System Evolution

What 64-Bit Taught Us

The Upward Compatibility Challenge

www.acmqueue.com



COMPLIANCE

Log on to www.acmqueue.com to download the ACM Queuecast to your computer or MP3 player.

Tune into Queuecasts: Discussions with the Experts as Kris Lovejoy, the CTO of Consul, discusses how and why companies should shift their compliance efforts to a governance approach that ultimately will better serve a company's needs.

Sponsored by IBM

acm queuecasts

discussions with the experts

Removes the hurdles between project planning and project implementation

DevPlan
Use DevPlan to define project deliverables

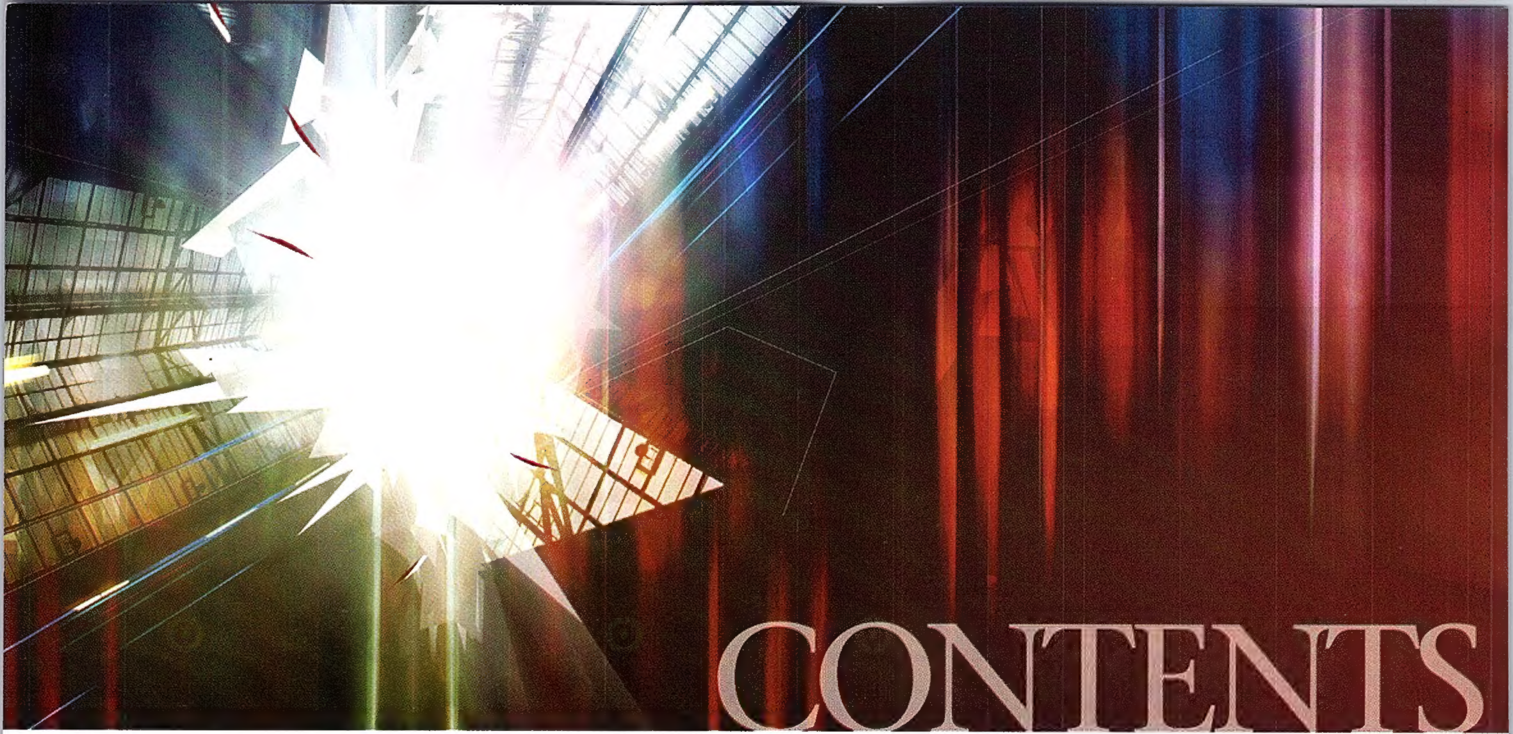
- Manage milestones, timelines, and resources
- Attach designs, specifications, and requirements to create a fully conceptualized product before development work begins
- Schedule events to co-ordinate resources, enforce deadlines, and drive deliverables

DevTrack
Use DevTrack to manage its implementation

- Manage implementation items, changes, features, defects, and project issues using an action-based workflow
- Track development work within the context of a project and associate work items to source code deliverables
- Run configurable reports that can be published dynamically to the web

TechExcel

www.techexcel.com | 1-800-439-7782



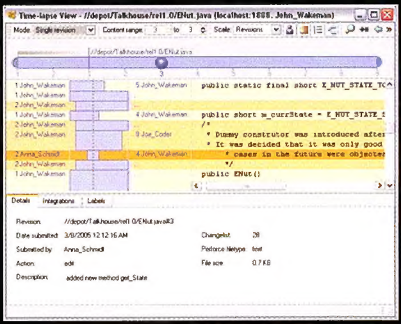
OCTOBER 2006 VOL. 4 NO. 8

rants: feedback@acmqueue.com

Perforce | *Fast* Software Configuration Management

Introducing Time-lapse View,

a productivity feature of Perforce SCM.


Time-lapse View - //depot/Talkhouse/rel1.0/ENut.java (localhost:1888, John_Wakeman)
Mode Single revision
Content range 3 to 3
Scale Revisions
//depot/Talkhouse/rel1.0/ENut.java
1 2 3 4 5 6 7 8 9
Details
Integrations
Labels

```
1 John_Wakeman      5 John_Wakeman   public static final short E_NUT_STATE_TC
1 John_Wakeman
2 John_Wakeman
1 John_Wakeman      4 John_Wakeman   public short m_currState = E_NUT_STATE_
2 John_Wakeman                       /*
2 John_Wakeman      9 Joe_Coder       * Dummy construtor was introduced after
                                      * It was decided that it was only good
2 Anna_Schmidt      4 John_Wakeman      * cases in the future were objecte
2 John_Wakeman                          */
1 John_Wakeman                       public ENut()
```

Revision //depot/Talkhouse/rel1.0/ENut.java#3
Date submitted 3/9/2005 12:12:16 AM
Changelist 28
Submitted by Anna_Schmidt
Perforce filetype text
Action edit
File size 0.7 KB
Description added new method get_State

Perforce Time-lapse View

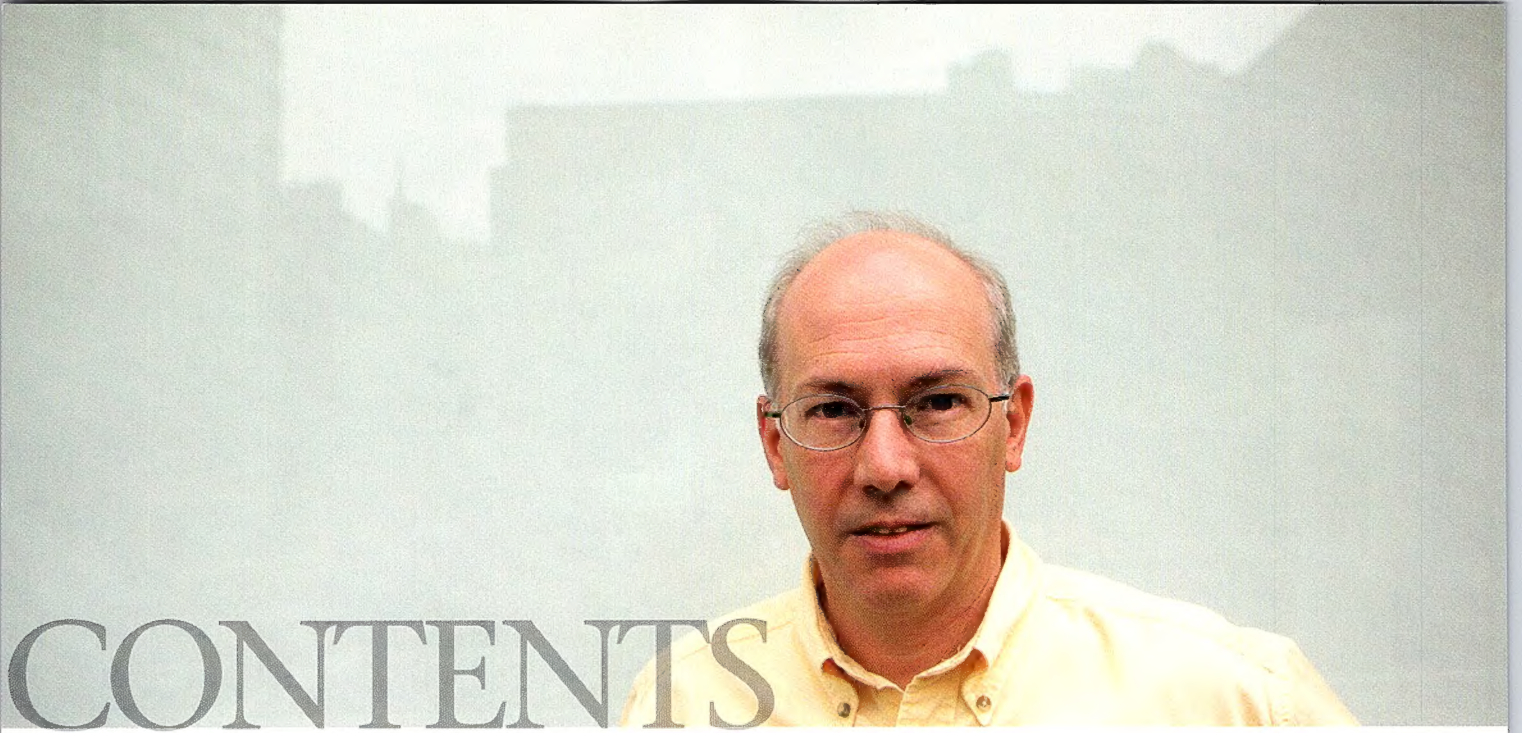
Time-lapse View lets developers see every edit ever made to a file in a dynamic, annotated display. At long last, developers can quickly find answers to questions such as: 'Who wrote this code, and when?' and 'What content got changed, and why?'

Time-lapse View features a graphical timeline that visually recreates the evolution of a file, change by change, in one fluid display. Color gradations mark the aging of file contents, and the display's timeline can be configured to show changes by revision number, date, or changeset number.

Time-lapse View is just one of the many productivity tools that come with the Perforce SCM System.

PERFORCE
SOFTWARE

Download a free copy of Perforce, no questions asked, from www.perforce.com. Free technical support is available throughout your evaluation.



CONTENTS

INTERVIEW

rants: feedback@acmqueue.com

Vinton G. Cerf Vice President and Chief Internet Evangelist Google	**Amy Wu** Computer Science Student Stanford University	**Benjamin Mako Hill** Research Assistant MIT Media Laboratory	**Maria Klawe** President Harvey Mudd College

ACM: KNOWLEDGE, COLLABORATION & INNOVATION IN COMPUTING

Uniting the world's computing professionals, researchers and educators to inspire dialogue, share resources and address the computing field's challenges in the 21st Century.

Association for Computing Machinery
Advancing Computing as a Science & Profession
www.acm.org/learnmore

acm queue

architecting tomorrow's computing

Publisher and Editor
Charlene O'Hanlon
cohanlon@acmqueue.com

Editorial Staff

Executive Editor
Jim Maurer
jmaurer@acmqueue.com

Managing Editor
John Stanik
jstanik@acmqueue.com

Copy Editor
Susan Holly

Art Director
Sharon Reuter

Production Manager
Lynn D'Addesio-Kraus

Editorial Assistant
Michelle Vangen

Copyright
Deborah Cotton

Editorial Advisory Board
Eric Allman
Charles Beeler
Steve Bourne
David J. Brown
Terry Coatta
Mark Compton
Stu Feldman
Ben Fried
Jim Gray
Wendy Kellogg
Marshall Kirk McKusick
George Neville-Neil

Sales Staff

National Sales Director
Ginny Pohlman
415-383-0203
gpohlman@acmqueue.com

Regional Eastern Manager
Walter Andrzejewski
207-763-4772
walter@acmqueue.com

Contact Points

Queue editorial
queue-ed@acm.org

Queue advertising
queue-ads@acm.org

Copyright permissions
permissions@acm.org

Queue subscriptions
orders@acm.org

Change of address
acmcoa@acm.org

ACM Headquarters
Executive Director and CEO: John White
Director, ACM U.S. Public Policy Office: Cameron Wilson

Deputy Executive Director and COO: Patricia Ryan
Director, Office of Information Systems: Wayne Graves
Director, Financial Operations Planning: Russell Harris
Director, Office of Membership: Lillian Israel

Director, Office of Publications: Mark Mandelbaum
Deputy Director, Electronic Publishing: Bernard Rous
Deputy Director, Magazine Development: Diane Crawford
Publisher, ACM Books and Journals: Jono Hardjowirogo

Director, Office of SIG Services: Donna Baglio
Assistant Director, Office of SIG Services: Erica Johnson

Executive Committee
President: Stuart Feldman
Vice-President: Wendy Hall
Secretary/Treasurer: Alain Chesnais
Past President: Dave Patterson
Chair, SIG Board: Joseph Konstan

For information from Headquarters: (212) 869-7440

ACM U.S. Public Policy Office: Cameron Wilson, Director
1100 17th Street, NW, Suite 507, Washington, DC 20036 USA
+1-202-659-9711–office, +1-202-667-1066–fax, wilson_c@acm.org

ACM Queue (ISSN 1542-7730) is published ten times per year by the ACM, 2 Penn Plaza, Suite 701, New York, NY 10121-0701. POSTMASTER: Please send address changes to ACM Queue, 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA Printed in the U.S.A.

The opinions expressed by ACM Queue authors are their own, and are not necessarily those of ACM or ACM Queue. Subscription information available online at www.acmqueue.com.

BPA WORLDWIDE BUSINESS

Reality vs. Perception

Charlene O'Hanlon, *ACM Queue*

October is upon us, the month that celebrates the struggle between perception and reality. Every October 31, children of all ages don costumes and masks in pursuit of receiving treats or playing tricks, trying desperately to fool everyone into thinking they're a witch or a goblin or a clown instead of a 10-year-old girl or a teenage boy or a middle-aged business executive.

I, for one, would rather celebrate reality. Don't get me wrong—I love Halloween and the joy it brings to all recipients of candy (myself included, since I always buy too much). But one of the most hazardous elements of life is perception, and often what is reality and what is perceived as reality are two dangerously different things.

I should have taken Latin in high school, because as I deal more and more with the issue of reality vs. perception I often find myself muttering a few of those well-known phrases that mean so much in certain situations. Most days, thankfully, it's more *carpe diem* than *mea culpa*. But in my quest to expand my multisyllabic arsenal, I've discovered a plethora of Latin phrases that fit well into my average days, including (in alphabetical order):

- *ad absurdum* (to the point of absurdity)
- *ad hominem* (appealing to feelings rather than reason)
- *advocatus diaboli* (devil's advocate)
- *de gustibus non est disputandum* (there is no accounting for taste)
- *dum spiro, spero* (as long as I breathe, I hope)
- *in vino veritas* (in wine there is truth)
- *velle est posse* (where there is a will, there is a way)

Judging from the popularity of some of the more common Latin phrases, I think our founding fathers of language had a lot of insight into the human condition and its associated ethos and egos. In dealing with the issue of reality vs. perception, however, sometimes insight isn't enough.

Anyone who has ever tried to balance a checkbook will tell you numbers can lie. Anyone who has been misunderstood in an e-mail will tell you that people will read what they want to read, not necessarily what you've written. And anyone who has been unfairly accused of an action will tell you that the accusation stings.

Still, no matter what the situation, one thing always rings true: Cleaning up after an unintended misperception can be downright messy—much like cleaning up bad code that was created to keep an outdated legacy system running years after it should have been retired. Unfortunately, in most enterprise environments, that scenario is more a reality than a perception.

Cleaning up
MISPERCEPTION (OR BAD CODE) IS A MESSY TASK.

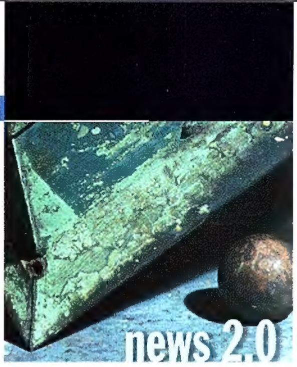
This month's issue deals specifically with that reality. How can a company plan ahead to enable its systems to grow old gracefully and be able to avert disaster when the processor starts smoking? How can it adjust its current operations to accommodate future advances in system technology? And when it comes to accommodating the advances, how fast is too fast? How slow is too slow?

John Mashey tackles the last set of questions and the (some might argue) disastrous results in his article, "The Long Road to 64 Bits." While the beauty of 64-bit architecture was too great to ignore, the constraints of implementing such a system on an epic scale proved to be a debacle for numerous companies, not the least of which were the chip makers, who lost revenue hand over fist, and the developers, who were building for a market that did not exist on a scale large enough. "Past decisions have unanticipated side effects that last decades and can be difficult to undo," Mashey writes.

Other articles in this month's issue deal with perceptions also, but on different scales. Damon Poole, founder and CTO of AccuRev, in his article, "Breaking the Major Release Habit," looks at the potentially time-saving method of agile development. Although the process can help developers save precious time (and save companies from the vicious cycle of major release patches), whether it can deliver on its promises in today's complex software development environment remains to be seen.

This month's issue is ripe with perception. The reality will make itself known soon enough. When it does, just follow the empty candy wrappers to find me. Q

CHARLENE O'HANLON can't be held responsible for what is said under the influence of too much sugar.



Taking a second look AT THE NEWS SO YOU DON'T HAVE TO

The Mobile Linux Challenge

Big news for the Linux community: Motorola plans to put Linux in 50 to 60 percent of its mobile phones over the next couple of years. The company already uses it in its popular ROKR E2 phone and will soon deploy it on a number of other moderately priced consumer models.

While Motorola's large-scale buy-in is a momentous step forward, its success could be limited by the many competing versions of Linux, which could fragment the market for applications that run on it. To address this problem, Motorola launched the Open Platform Initiative, which aims to standardize the operating system among the growing list of handset manufacturers that use it. Two other groups that promote mobile Linux standards, Open Source Development Labs (OSDL) and Linux Phone Standards Forum (LiPS), recently announced plans to collaborate. This could make it difficult for the Motorola effort to gain traction.

One thing is clear: Without a standardized platform, it will be difficult for software vendors to create the large, vibrant ecosystem of mobile Linux applications essential for Linux's success.

WANT MORE?
http://news.com.com/Motorola+dialing+up+mainstream+Linux+phones/2100-7252_3-6106065.html

Google not into Googling

In a somewhat uncharacteristic move, Google has written to media organizations requesting that they refrain from using the now widespread verb form of the company's name: "to google." In the letters, the company included examples of appropriate and inappropriate usages of its trademark, worded in a playful vernacular that one might expect of the independently minded tech juggernaut.

Why this apparently serious concern over googling? One reason could be the perception that generic use of the term might dilute Google's brand image. Or perhaps the company is concerned about efforts by other companies to piggyback on the brand (see TV ad with screen shot: "Don't take our word for it, google 'Pontiac' to find out!"). Another possibility is that in light of Google's ever-expanding line of products and services, the company wishes to avoid being associated with just search.

Despite its efforts, the company is unlikely to stop the spread of googling. Both the Oxford English Dictionary and Merriam-Webster Online include the verb "google" (albeit with full credit to the word's trademark etymology), and short of any Orwellian scenario, stemming casual use is impossible. But beyond considerations of the feasibility of Google's crusade, many question its advisability. After all, isn't it a mark of success to be perceived as the final word in search?

WANT MORE?
http://seattletimes.nwsource.com/html/businesstechnology/2003178630_google06.html
http://news.com.com/Google+wants+people+to+stop+googling/2100-1030_3-6106479.html

Keeping Online Video Legit

As video sites such as YouTube grow in size and popularity, concerns are mounting over copyright infringement. With thousands of files being added per day, it's becoming harder and harder to police, so new technological solutions are springing up to combat the problem.

One solution is video fingerprinting, which extracts unique bit streams, or fingerprints, from video files and loads them into a database. Software then scans incoming video files and matches them against the database. A match means someone is trying to post copyrighted material and the file will be blocked. Video site Guba is already using its homegrown fingerprinting software and Philips will soon release its commercial solution.

It's still too early to tell how effective the technology will be at fighting copyright infringement. Pirated video often is cropped or converted to different compression formats, which could be difficult for the software to detect. And even if the matching process is fast and accurate, the technology must meet the original challenge of keeping up with the endless stream of copyrighted video content. Then again, it need not be perfect, just good enough to protect these sites from being seen as safe havens for pirates.

WANT MORE?
http://www.technologyreview.com/read_article.aspx?id=17343&ch=biztech Q

What's on Your Hard Drive?

The name of this department, "What's on Your Hard Drive?", suggests software that's installed locally on your desktop or laptop. With the amount of software that now runs on remote servers accessed through a Web browser, however, we're expanding our scope to include any of these thin client tools or services that you might find useful. So stop by http://www.acmqueue.com and let us know what's on your hard drive…and what isn't.

Who: Tim Butler
What industry: Technology vendor (software, hardware, etc.)
Job title: Software engineer
Flavor: Develops on Linux, Solaris, AIX, HP-UX for Linux, Solaris, AIX, HP-UX
Tool I love! DTrace. While perhaps not typically considered a dev tool, it is to those of us in performance analysis. The global integrated view from Java bytecode, to native user space, to kernel, to device driver, is unparalleled.
Tool I hate! Visual Studio. Its complicated bloat makes automated audited builds nearly impossible. All dev tools need to be version controlled to ensure repeatability, and VS makes that impossible.

Who: Christopher Ness
What industry: ISP/Telecommunications, energy, cable, utilities
Job title: Project designer
Flavor: Develops on Windows for Windows Mobile
Tool I love! Subversion. Subversion and the revision control it offers has saved my work more times than I can count. I would be lost without the history of my code that it provides.
Tool I hate! The mouse. I hate having to take my hands off the keyboard. I wish developers would think more about power users and shorten the keystrokes to do common tasks, allowing me to work as fast as I can think.

Who: Ernie Stewart
What industry: Manufacturing (noncomputer)
Job title: Application architect/business analyst
Flavor: Develops on Windows for Windows
Tool I love! Visual Studio. Everything I need to develop professional business solutions fast is under one roof—editor, compiler, debugger, help system, database/query tools, wizards. My favorite feature is IntelliSense, because with it I don't have to remember identifier names or function syntax. I like being told I've made a mistake before compiling and testing.
Tool I hate! NetBeans. It's hard to understand what it's trying to abstract. I often prefer to use external HTML and resource editors rather than installing and configuring plug-ins. With Visual Studio it's ready to go right out of the box.

Who: Brian Donovan
What industry: Technology vendor (software, hardware, etc.)
Job title: Software engineer
Flavor: Develops on Windows for Windows, GNU/Linux
Tool I love! EditPlus, a beautifully simple text editor for Windows. Once it gets code folding (in beta now), it will be perfect. EditPlus is totally color customizable and has syntax highlighting, line numbers, visible whitespace, and UTF-8 support. It feels infinitely customizable. SciTE (Scintilla Text Editor) comes close (and already has folding), but needs intelligent word-wrap and a prefs UI.
Tool I hate! Eclipse. It is frustrating because it has so much potential (because of the plug-ins available), but, unfortunately, the prefs UI is a mess and there's no soft wrap (though someone has taken this on as a Summer of Code project), which is a killer for me.

Saddle Up, Aspiring Code Jockeys

A koder with attitude, KV ANSWERS YOUR QUESTIONS. MISS MANNERS HE AIN'T.

This month KV offers advice to readers considering careers as full-time coders. He suggests that new programmers learn everything they can by reading magazines, journals, and Web sites. And what better place to engage in such learning than this very column? It might be a while before we publish our deluxe, hardcover *Kode Vicious Compendium*, so until then we suggest you visit http://www.acmqueue.com, where we've archived all past KV columns for quick and easy reference.

Dear KV,
I am an IT consultant/contractor. I work mainly on networks (I'm a Cisco Certified Network Associate) and Microsoft operating systems (Microsoft Certified Systems Engineer). I have been doing this work for more than eight years. Unfortunately, it is starting to bore me.

My question is: How would I go about getting back into programming? I say "getting back into" because I have some experience. In high school I took two classes of programming in Applesoft BASIC (archaic, I know). I loved it, aced everything, and was the best programming student the teacher ever saw. This boosted my interest in computer science, which I pursued in college.

In college, I took classes in C++, Java, and Web development (HTML, XHTML, JavaScript). I did great and had fun. For various reasons, I ended up leaving college and becoming a network administrator, and for the past eight years have been doing this and that and everything else IT-related. But I haven't been programming. The extent of my programming work experience is with MS Excel macros (yuck!) and basic VB coding in Microsoft Access.

So how could I start becoming a programmer? Visual Studio 2005? Java? Eclipse? I enjoy self-learning and have found that achieving certifications gets my foot in the door. Is there a particular suite that I could get certified in to get my newly desired career going?

Thanks in advance.

Jonesing for a Job

Got a question for Kode Vicious? E-mail him at kv@acmqueue.com—if you dare! And if your letter appears in print, he may even send you a Queue *coffee mug, if he's in the mood. And oh yeah, we edit letters for content, style, and for your own good!*

Dear Jonesing,
Wait, you've gotten so good at your job that you're bored? Why not take up golf, or painting? Perhaps download more "content" from the Internet! Why on earth would you want to give up a job that you know just to become a programmer?! Don't you know that programmers put in long hours to meet impossible deadlines, all to make a bunch of rich guys richer?

OK, if you've gotten this far, then I guess maybe I should actually answer your question. There are as many answers to your question, of course, as there are programmers. How you go about moving into a coding job depends on a few things. The first is, "What do you like to do?" There is no point in working hard to learn a language or a system if you don't enjoy it. You'll just wind up right back where you are now, wanting to do something else. Find the kinds of problems you like to solve, then see how they're solved today, and see if that's the kind of thing you like to do.

Since you say you already have a computer science background, I don't really see much reason to go for any special certifications. I have never seen the point of most certifications as they certify only that you can pass a test, not that you can think about problems, which is the more important skill. I also don't think you should worry about which set of tools you use just yet. There are other things to work through, like step 2.

The second step, and be thankful that there aren't 12 of them, is to pick a project. I can't learn anything unless I have a project around what it is I want to learn. Try to choose something you think you can actually do. "Write an operating system," though it's a fun goal and actually possible to do, is likely not the right place to start. Working on an open source project such as FreeBSD, Apache, or Open Office might be another way to get started. Find something you need to use on a relatively frequent basis and try to extend or fix it. Most open source projects have long lists of open bugs. Pick a few of those and try to fix them and submit patches to the maintainers.

Finally, take every opportunity you can to learn about your new area. That doesn't mean spending lots of money—or conning your employer into spending lots of money—on classes and conferences. Of course, if you find a class in Hawaii and your employer is willing to send you, well, I can't argue with that but I can't call it learning either. Instead, regularly read the journals, magazines, and Web sites that cover your newfound area of expertise.

Once you think you have what it takes to start your new career, look at entry-level positions that will allow you to learn more. Be prepared to take a pay cut because moving from eight years of experience in network administration to entry-level programming isn't likely to be a big monetary win, but then you're doing this for the fun and challenge, right?

KV

Dear KV,
Because of performance needs, my team is reworking some old code to run multithreaded so that it can take advantage of the new multicore CPUs that are now shipping in high-end servers. We're estimating that it will take at least six months to break down our software such that it will be granular enough to run as multiple threads and to implement all the proper locking and critical sections. I came across an old presentation online, "Threads Considered Harmful," and was wondering what you thought of it. The paper was written long before we had multicore CPUs. At the time there were few commercial SMP (symmetric multiprocessing) machines, so perhaps it didn't make sense to go to all the bother of writing threaded code then, but now things are different. Have you heard of this paper? Do you think it is still valid?

Hanging by a...

Dear Hanging,
John Ousterhout's warning (Why Threads are a Bad Idea [for most purposes], 1996 Usenix Technical Conference) is as important today as it was when it was written, not because times and technology haven't changed, but because, alas, people haven't. Most people seem to decide to create multithreaded code for the reasons you state here—that is, to get a supposed performance boost from it. These same people never seem to bother to measure their code or see if it is even practical to run it in multiple threads. They just start slicing away at the code in the vain hope that if they have enough threads, suddenly, as if by magic, their code will run faster.

Longtime readers of KV will know that I do not believe in magic bullets. Waving a wand labeled "threads" over your code is about as likely to make it run faster as is sacrificing a chicken. At least you can eat the chicken when you're done with it, which is more than I can say for your code. In actual fact, threading your code may make it run slower, because poorly written threaded code is often slower than poorly written nonthreaded code. The locking primitives required to get locking right are nontrivial and can, if improperly used, slow your code down, as it all blocks on the same lock or, even worse, can introduce subtle bugs.

Another problem with threaded code is that the tools used to debug it remain primitive. Although most debuggers now claim to handle threads properly, this is not always the case, and you really don't want to be debugging your debugger while debugging your code.

One last thing that people miss in their rush to thread their code is the support they get from libraries that they link against. If your program requires that the libraries it uses be multithreaded, then you may be in for a shock when you realize that some of them are not thread-safe. Using non-thread-safe libraries in your thread-safe program is going to cause you no end of trouble.

Given all of this, should you still continue to thread your code? Maybe. First you need to understand the trade-offs and see if the job the code does is amenable to being multithreaded. If the code has several components that can operate completely independently, then yes, multiple threads can be a boon. If, on the other hand, the components all need to access a small shared section of data all the time, then threads will get you nowhere. Your program will spend most of its time acquiring, freeing, and waiting on the locks that protect the shared data.

So, unless it's really a win, and you have thought about it a lot, I would try not to get hung up in threads.

KV

KODE VICIOUS, known to mere mortals as George V. Neville-Neil, works on networking and operating system code for fun and profit. He also teaches courses on various subjects related to programming. His areas of interest are code spelunking, operating systems, and rewriting your bad code (OK, maybe not that last one). He earned his bachelor's degree in computer science at Northeastern University in Boston, Massachusetts, and is a member of ACM, the Usenix Association, and IEEE. He is an avid bicyclist and traveler who has made San Francisco his home since 1990.

_INFRASTRUCTURE LOG

_DAY 15: This project's out of control. The development team's trying to write apps supporting a service oriented architecture, but it's taking forever. Gil's resorted to giving them all coffee IVs. Now they're on java while using JAVA. Oh, the irony.

_DAY 16: Big crisis—we've just run out of half-and-half!!

_DAY 18: I've found a better way: IBM Rational. It's a modular software development platform based on Eclipse that helps the team model, assemble, deploy and manage service oriented architecture projects. The whole process is simpler and faster, and all our apps are flexible and reusable. The software we write today will be the software we use tomorrow. :)

_The team says it's nice to taste coffee again, but actually drinking it is sooo inefficient!

Download the IBM Software Architect Kit at:
IBM.COM/**TAKEBACKCONTROL**/FLEXIBLE

A Conversation with David Brown

Photography by John Abbott

The nondisruptive theory OF SYSTEM EVOLUTION

This month *Queue* tackles the problem of system evolution. One key question is: What do developers need to keep in mind while evolving a system, to ensure that the existing software that depends on it doesn't break? It's a tough problem, but there are few more qualified to discuss this subject than two industry veterans now at Sun Microsystems, David Brown and Bob Sproull. Both have witnessed what happens to systems over time and have thought a lot about the introduction of successive technological innovations to a software product without undermining its stability or the software that depends on it.

David Brown began his career in computing with the disruptive innovation brought about by the microprocessor generation: first on the SUN project at Stanford University and then as a founder of Silicon Graphics. He has now worked for more than 14 years on the Solaris system at Sun, where nondisruptive innovation (evolution and sustainability) is critical. Both the length and breadth of Brown's career give him a unique perspective on systems and how they evolve. Brown led the Application Binary Compatibility Program for Solaris and discusses how new functionality is introduced in that system without breaking existing applications.

Interviewing Brown is a legend in the field, Bob Sproull, a Sun Fellow and head of Sun Labs. Sproull is well known for his early work in computer graphics and the seminal work, *Principles of Interactive Computer Graphics* (McGraw-Hill, 1973), which he wrote with William Newman. Sproull was also a member of the pioneering team at Xerox PARC that designed the first personal computer, the Alto, and its operating system. Sproull's comprehensive systems expertise makes him deeply familiar with the key problems inherent in their evolution.

BOB SPROULL To begin, let's hear about some of your roles in the Solaris engineering group at Sun.

DAVID BROWN I arrived at Sun in 1992, around the time of the change from SunOS 4 to Solaris. At that time we were beginning the Systems Architecture process, which I was hired to help design and get started. This was because we observed a couple of basic things.

One was that we now had several hundred engineers concurrently developing projects to put into Solaris, and there was a question about how we were going to do traffic control on all that and preserve the integrity of the system.

The second was this Homer Simpson "Doh!" moment when we found that our existing customers weren't terribly excited about going to the new Solaris 2 system because it didn't run a lot of the stuff they already had. This derived from the fact that in going from SunOS 4 to SunOS 5 we initially cut over a little bit too carelessly: We did a largely incompatible "big bang" at first.

My initial day-job in 1992 consisted of thinking about what our engineering review processes needed to be to get this whole System Architecture effort off the ground. We realized that we needed to be very careful about how we made changes to the system. One basic idea was to have every new project going into the system reviewed from an architectural perspective by the senior engineers. Another was to track the interfaces we were putting into the system that people were going to come to depend on, so we could preserve those and not disrupt applications or other layered products built on top of them. A big focus was defining what every new project's "interfaces" would be. This was a bit of a challenge at the outset because many of the engineers—including the senior engineers like me who were supposed to be thinking about and reviewing this—often didn't understand what *interface* actually meant: interface offered to whom, and for what purpose?

BS Could you summarize what you think the issues are that have to be balanced between the vendor (the operating system vendor, implementer, evolver, changer) on the one hand and the customer who wants to use it for his or her application or running a business on the other hand? What are the forces that animate changing the operating system in the first place?

DB The most obvious animating forces come from the customer. They have these things called applications that they use to run their businesses. They've made big investments in developing and deploying them, and they're very interested in keeping them up and running. This concern about not disrupting their existing applications extends to the layered software upon which those things might run—and all in its compiled form.

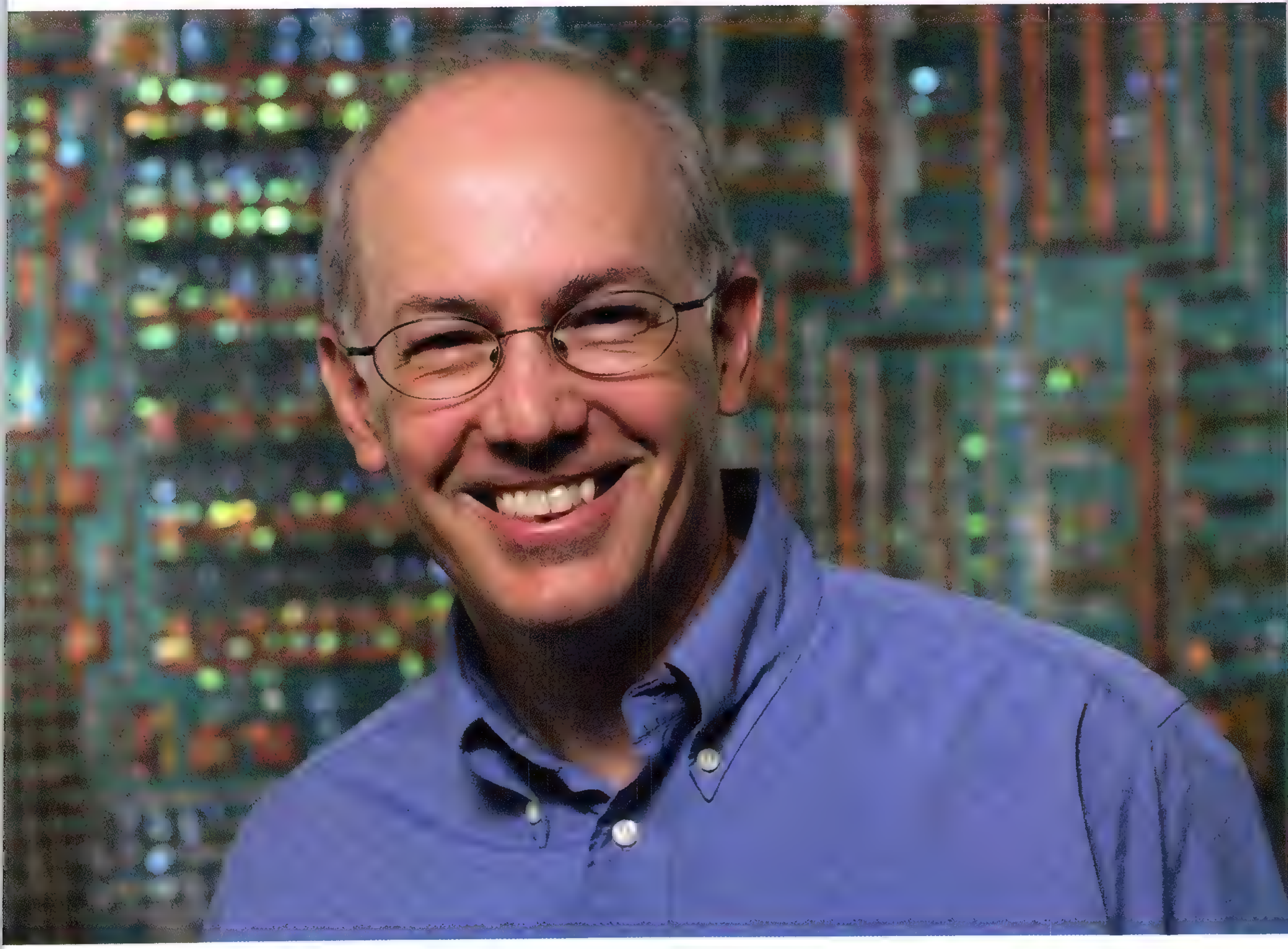
In other words, they're not just saying, "Oh gee, can we recompile the source code to get another working binary?" Rather, they are saying, "Gosh, I've already deployed thousands of these things across my enterprise, and I want those existing running binaries to continue to work without change when new releases of the operating system (or other parts of the system software) come out."

BS Part of the problem is the customer wants the benefit of all of the advances, but no disruption of their world. They want the faster hardware, the bigger memory, the bigger, faster disks, the automatic file mirroring. They always want more performance.

DB Yes that's also true. But on the customer side, rule number 1 is: "If you undermine what I've already built, I'm not interested."

End users and customers are very attached to what I refer to as the "problem set." They're very invested in solutions that help them go after their particular business problems. On the systems software (vendor) side of technology, we're very absorbed by the "solution set"—all these technological opportunities that are before us. The big motivator for us is to bring these new benefits out to our customers (and thus attract them to buy our new systems). Although it's clearly also attractive to the end users and customers as you point out so correctly, I tend to view this as being more vendor-driven.

But these two things—new features vs. stability of existing ones—tend to be in tension. In order to push



new features into the system—and this may be the crux of what we're focused on here—you have to be very careful about the way in which you deliver them. They can't undermine any of the existing commitments that you've made in previous versions of the system, because people have already built applications or other layered software upon them that needs to continue to run.

BS Let's move on to talk about the techniques, both technical and organizational—or just structural—that you can use to manage interface evolution, or just evolution generally. I'll start with a simple question: What do you mean by backward compatibility? And what do you mean by sustainability?

DB Backward compatibility, or what we usually call "upward compatibility" in terms of the way we build the system, means at a given point in time—let's say Solaris 2.1—the system has a bunch of features that can be exploited. These are exposed through some interfaces: the means of getting access to them, so that applications can build on them and use them. Upward compatibility means that in the next release of the system, Solaris 2.2, all of these same capabilities are still present, and more specifically, they're accessible in the same way as they were in Solaris 2.1. You might be creating new features and new capabilities, but these are in addition to what was there before.

BS Moreover, you don't have to recompile existing applications if you don't want to use the new features.

DB That is a very important point. The focus is on existing applications in their binary (already compiled) form. This means we're interested in the applications' runtime (as opposed to build-time) interface to the system.

When Sun was a technical desktop company—in its early disruptive days—most of the customers were scientists and engineers who were happy as long as they could recompile their source code for the next release. But in the evolution to Solaris 2, when we shifted our focus to sell these systems to enterprise commercial customers, things changed. What you find is that it's not just a couple of guys at Lawrence Livermore Labs who are writing their own code and retreading that stuff all the time anyway. Instead, it's these guys at Morgan Stanley with this enormously complicated bond trading system that has taken years to build, and they got it running on Solaris 2.6. But now we've come out with Solaris 2.8, which supports the blazingly fast new hardware (or whatever): It offers some various other advantages of scale, performance, or robustness that make it attractive, and we want them to be able to use that.

It's not so much that the Morgan Stanley guys don't want to have to recompile their application, but the recompilation to get a new binary also implies that they have to redeploy it to all these places where it's already running. They want the existing application, in its already-compiled form and as deployed, to continue to run when they upgrade to the new version of the operating system. ("Build Once, Run Forever" was a play on the Java marketing slogan that was coined to describe this.)

BS How do you achieve that? Could you also talk a bit about shared libraries and their naming?

DB The key question is: How are we going to characterize what the existing deployed applications (and layered infrastructure) depend on? One way to look at this might be: What are the ways in which an application could be broken? There's a subtle difference between those two questions that's very important. Trying to characterize all the ways in which any application could be broken is really daunting. There are just too many ways in which something could be undermined.

So, an important principle for me in approaching this was to forego thinking about complete solutions (in order to deal with any possible way in which any application could be undermined), and instead to ask: What are the things that every application must do to get access to the functionality in the system? Then make those primary, necessary pathways as robust as possible—really look at what happens there and ensure that the common things that every app has to do will work well.

A primary matter is the way that applications get access to the system's functionality. In the earlier days, it was by making system calls. Increasingly, however, most of the application-relevant functionality is wrapped up in these things called libraries: collections of related functions such as I/O-related functions, thread management, process management, or file access. The operating system's kernel—the protected lower-level piece of the system—tends be much less directly relevant to the application. It offers these more primitive low-level facilities such as the virtual memory abstraction and process scheduling and so on, whereas the applications are much more interested in things like: open a file; read and write the file; do some I/O. Those are offered by the system's libraries. That's the primary vehicle whereby applications get access to the system.

A further, particularly important feature of Solaris 2 (and most other systems of its generation—this is kind of a late '80s, early '90s technological shift) is dynamic linking. In Solaris, the way that applications bind to libraries (and the interfaces within them) is through linking at runtime. In the old days the application's linkage to the libraries was done at compile-time (what we call static linking). This essentially copied parts of the implementation of the library in question into the application binary. The application's runtime binding interface was then the system calls (those traps that these bits of library implementation used to access the kernel's services).

A really important consequence in Solaris 2 (and more recent systems of its ilk) is that the runtime binding is between the application and the system's libraries. Effectively it's the surface that's exposed by all of the libraries in the system that now constitutes the application's runtime interface—what I like to call "the Solaris Virtual Machine." As it happens, the executable code in each of the system's libraries is also shared by all applications that use it: There's only one copy of libc, for example.

BS But as you point out, the key aspect is that it's dynamically linked, not just shared.

DB The point really is that since they are dynamically linked libraries (since the binding between the app and the libraries happens at runtime), the application's runtime interface is at a much higher level than what was happening before. Now (say, circa 1990 onward) you have to realize that the entire collection of the system's libraries—and not just the kernel—represents the runtime system in total.

BS So this allows you as the systems developer to honor and maintain the contract at the library interface, which is exactly what software developers want.

DB Hopefully, that's what they want. Sometimes there's some confusion about that. Part of the sociological transition was to get application developers to realize that they do not want to bind the implementation of the C library into their applications. What they want to do is bind a dependency on the services that the C library offers through its interfaces. Then in Solaris we can actually improve the implementation of the C library in subsequent releases of the system (whether to increase its performance, maybe multithread its implementation, and so on), and your existing application binary gets all the benefits of that when running on a later system release, without making any change at all.

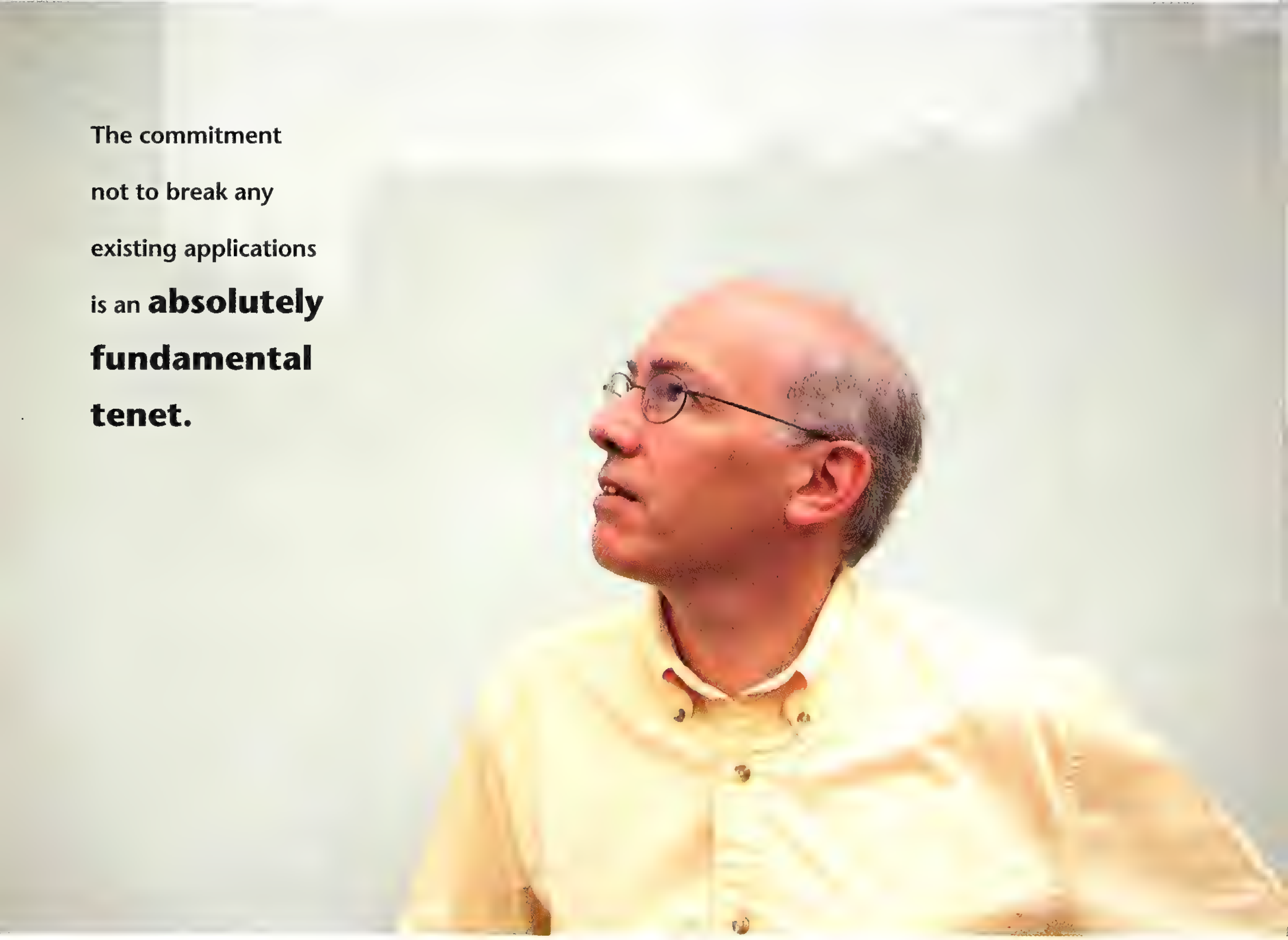
BS A classic example of the benefits of late binding.

DB Right. What you begin to realize is that the point at which runtime binding is occurring for applications of this class is not about making traps to get into the kernel—the traditional system call interface. What you're now doing is making these runtime bindings one layer up: on top of these libraries. Now in Solaris, the thing that all applications must do is make dynamic bindings to these library interfaces (C language binding to library global symbols in our case). If you can characterize that surface to a first approximation, you've defined the application runtime interface to the system, and then this becomes the contract that you want to define and maintain clearly.

BS I'm guessing that as the number and complexity of these interfaces grew, it was important to have some tools—both for you and for customers of these interfaces—to help understand the inventory of interfaces that they were depending on (perhaps down to specific versions), not only to track down problems but also to understand the finer structure of the system that they were building and deploying.

DB Yes, the need for tools to inspect interface dependencies is a very important point. But I think there are a couple of stages leading to this. The first was recognizing that the system libraries are the important dividing line. But then you have to get everybody's attention: Obviously, you have to get the attention of the operating system engineers who define and stabilize that boundary. But, also, anybody who's developing applications has to realize what the site of the contract is, too. If they don't know where the surface is, then they don't know how to confine themselves to using the ABI (application binary interface) it offers. Now you can get to the tools to help look at all that.

The commitment not to break any existing applications is an **absolutely fundamental tenet.**



Excuse me for saying this because it sounds so obvious, but you'd be surprised how few systems do this well.
BS I think it's easy for this to get obscured, because often I'm not just using Solaris's interfaces. For example, today I might import Perl over the Web and build applications on top of that. Whether I recompile it or not, I have no idea what Solaris interfaces Perl is using, unless there are tools that help me ferret out, or define, that surface. As I use bigger and bigger packages that are prepared by others, my visibility into this whole issue of managing change and evolution at the operating system's interfaces gets reduced.
DB Yes, this level of indirection represented by middleware seems confounding. It's a topology consideration that's part of the complexity question you raised. Proper inspection tools address that, too, but first let me return to the point you made about scale. The number of interfaces that represents the application runtime interface (or ABI) in a system such as Solaris may be on the order of 10,000 (say 200 libraries with an average of 50 functions each)—a somewhat daunting scale.

Given the scale of the interface in a serious system (whether it's IBM'S MVS, DEC's VAX/VMS, or Sun's Solaris), you must have some mechanical way to examine an application (or a piece of layered middleware) to decide that it stayed within the bounds of the safe and stable interfaces that the system was offering. There was this standing joke: Some developer would say, "Hey, how was I supposed to know what I could and should use, versus what I shouldn't?" Then this "RTFM" retort would sally forth: "Read the [fine] manuals." But that's absurd. You can't reasonably expect anybody, by reading the manuals, to make an assessment about whether or not what they coded to was OK. There are just way too many interfaces in the system for that.

In practice, it's hugely important that you have some mechanical way of examining an application: to see what it's using—and to decide whether that's OK or not.
BS By interface, do you mean a particular function call or do you mean a group of related function calls?
DB I mean an individual entry point. Each library (group of related function calls) is also an important interface since the library names are used by applications to get to the individual functions in question. But libraries are the containers, and that's not sufficiently fine granularity. It's necessary to inspect whether applications restrict themselves to the set of shared objects that we said were OK (libraries such as libc, libthread, libaio, etc.) but inadequate. More specifically, they must restrict themselves to just those interfaces intended for applications. Applications must not use any internal or system implementation interfaces within system libraries. Admittedly, in Unix-derived systems, it's an artifact of the C programming language and its limited interface scoping capabilities that these are visible to applications.
BS I think the point you're making is that to be able to understand the dependency on an entry-point-by-entry-point basis is extremely useful in figuring out how changes may propagate to influence you.
DB Yes, and that's because the system can be evolved not only by the introduction of new libraries, but also by the addition of functions to an existing library.

BS Let's switch from compatibility to focus instead on evolution. How do you introduce change in a controlled way?
DB There are two aspects to this. One is technological, and the other is more conceptual or sociological. Indeed, I think the latter—the principles and adherence to them—is really the bigger thing.

We came to realize pretty early on that when you're making changes to the system, if you do something that takes away existing functionality or makes some other incompatible change to the stuff that's already there, that has negative impact on your customers because it's going to undermine some applications that have already been camped there.

The commitment not to break any existing applications is an absolutely fundamental tenet. The understanding is that strict upward compatibility is a constraint, not just a goal. You cannot break anything that has already gone before.

This translates directly to how you offer new functionality (whether or not you define new interfaces to deal with it). First, it must be engineered so that it won't break any existing interface or offer that's part of the contract (ABI). Next, you must really think about the public access points to this new functionality, because you're going to be committed to maintaining that functionality in a strictly upward compatible fashion subsequent to its introduction.

So, there's an initial couple of principles: Maintain strict upward compatibility from one release to the next, and define the application interfaces clearly so you don't allow any applications (or layered software products) that use them to be broken in a later release of the system.

Both of these things are easy to say, but they can have pretty broad implications in terms of what you must actually do when you're making changes to the system. If you're introducing new functionality and it's completely independent of anything that's in the system right now, it's pretty straightforward what you do: Add new functions to a library, or add a new library containing the functions. It's an additional component—new stuff you can use. The engineering technique is really obvious.

But then you can get slightly trickier circumstances when you impart change that is a bit more pervasive. For example, in going from uniprocessor to multiprocessor platforms, you would like to introduce the ability to have multithreaded applications. Now you have a change that is going to affect a lot of the existing interfaces that are already in the system.

It adds new capabilities or semantics that are going to be exposed through some existing interfaces. For example, the Unix system has this important interface called `fork()`, which relates fundamentally to the process model. It's the basis of concurrent execution in the system. Prior to multithreaded applications, the fork interface meant: Make a copy of this process and then run that concurrently with this process. But what does `fork()` mean now, in the case of multithreading, where you have multiple concurrent threads of execution running in a single process?

It could mean copy the process for all its threads of execution, or it could mean copy the process to execute just this one current thread of execution. There are some decisions to be made about what the new semantics of this existing interface are going to be, in light of this new pervasive feature change. You also have to ensure that the introduction of the new semantics in no way affects the abstraction that single-threaded applications were relying on in the past. The introduction of 64-bit interfaces is a similar conundrum (although perhaps worse still because it impacts the semantics of fundamental data types at the language level).

Those are the places where things can get a little trickier. A little bit more cogitation may be required while doing the engineering to ensure that you've thought through all that's already out there and what it depends on and how you want to impart this new functionality.

BS When you introduce something brand new, you may not get it quite right, or put slightly differently, it may evolve relatively rapidly compared with things that are tried and true. The question I have is perhaps more sociological than technical. Are there any techniques that have worked for introducing new things and letting you, the offerer, change them more rapidly earlier on until you get them right?

DB In systems that are at a more mature point in their evolution, a huge amount of their functionality has already become quite stable. We might just observe much of this existing stuff and simply declare that it's stable and we're not going to mess with it.

But the sharp edge of this equation is just what you described: When you've got this new technology you want in the system—but you haven't quite got your head around what it should look like, or perhaps even what features you think are going to be wanted—there can be a lot of pressure to get this experimental interface out there to gain some experience. You want to put this stuff in the system for developers to play around with and build test applications on top of so you can get some feedback. But you're not committed to these interfaces yet, so it must be clearly distinguished somehow.

In Solaris quite early on, we came up with this idea of Interface Taxonomy. It's a classification that essentially defines the intended scope of use of an interface: Is it intended for use by third-party applications; or for use by other parts of the Solaris implementation; or is it internal to the implementation of this particular project? We call the various levels in this taxonomy the interface's commitment level or, analogously, its stability level. These levels reflect the kind of change that can or cannot be made to the interface when the system is changed. Must it be maintained in a strictly upward-compatible way, or could it be changed in an incompatible way?

For example, we have the Public level (which indicates that application software may use the interface), on down to commitment levels such as Consolidation Private and Project Private (to reflect interfaces that are to be used only within the implementation of a particular subsystem or even a particular implementation project). Each interface introduced by any engineering project that will add to or change the system must be defined in this way.

Simply stated, as the exposure and use of an interface becomes broader, the degree to which we must maintain its stability becomes much greater. Of course, primary interest is with those interfaces labeled Public, because that's where third-party applications and other layered software are allowed to camp.

After giving some more thought to the full software lifecycle—including both the emergence of new functionality and the obsolescence of really aged stuff—we introduced commitment levels called Experimental and Evolving. The idea was that such interfaces would not be part of the guaranteed safe and stable terrain; they were for developers to experiment with. The Evolving level was the halfway house to Public: We would make our best effort to maintain compatibility with those, but it was too early in the lifecycle of the technology to be confident that this was what we really wanted.

For Solaris, this definition of interface scope is really a primary aspect of the system's software architecture, and the basis for how we evolve the system gracefully. It says who/what we expect can use any given interface, and what constraints are therefore imposed on us when we make changes to the system as a result of that. Ultimately, it has a lot to do with defining the rules of engagement with people who are building stuff on top of our system. It says, "Hey, third-party application developers who might resell this stuff to other end users, thou shalt not resell something that's built on top of these Experimental interfaces because that's not guaranteed to be safe." The trouble with that is ISVs can't get the in-the-field experience with customers.

BS Interfaces are the modularity or the construction techniques that we have in the software engineering world, and interface technology has progressed beyond where it is with C global symbols, for example. We now have languages that type-check across interfaces, but it seems to me that there are still a lot of things that are implicit in interfaces that we don't check and that are implicit in the interface contract that we don't check for invariance.

One of my favorite bugaboos is performance: that certain functions are assumed to be fast, while for others, it's OK to be slow.

For example, in libc, if I get a character, I assume that has to be pretty fast. If I read a million bytes, I assume that can be slower, and almost all the performance contract is implicit. I've never seen even a comment, let alone a piece of descriptive syntax, that is actually interpreted in any way that commits an interface to certain performance properties. Yet the correctness, perhaps not in the functional sense but in the merchantability sense of our applications, depends completely on a performance contract that isn't broken from one version of the system to the next.

If you had a magic wand and could add something to interfaces that could be checked, whether dynamically or statically, what would give you the most benefit in terms of preventing the kind of breakage or backward compatibility problems when you evolve the system independently of a customer evolving an application?

DB There's this question about the way we've gone about defining the interfaces, which I described unabashedly as an idiot-simple enumeration of the application contact points. It's the set of knobs that you're allowed to grab onto, and we give you an electric shock if you grab a disallowed one. But it's much more challenging to characterize some of these usage-related properties of the interface. I like to use the word *semantics* for that, and performance is one good example. There are some expectations about how responsive each interface is, and that is in no way characterized in any interface specifications or definitions that we have. Many other behavioral aspects of the interface are equally unspecified in any rigorous way.

Currently, things such as performance and other semantic integrity questions are assessed by pre- and/or post-integration testing. We have something called Perf PIT for the former. These happen after project implementation and are outside the domain of interface definition. There's a bunch of performance benchmarks, some of which are industry-standard things, such as TPC-C (the

transaction processing benchmark of the Transaction Processing Performance Council). We just run these tests as part of the system's build process, whether before or after the project has been integrated, but typically after it has been developed and certainly before release. Then if there's a performance regression, we engineer like mad to fix it.

The provocative question was how might we drive these semantic stipulations back into the interface definition practices? When I was involved in technically leading the ABI program, one of the big decisions I made was about how much energy and effort to put into interface definition and interface-related practices versus other mechanisms. One of the really big watersheds for us was the huge gulf between syntactic definition of the interface and the semantic definition of the interface.

Characterizing the semantics of the interface and testing was going to be such a huge, ominous, burdensome sort of thing. I wondered how we were going to get all of the engineers in the company who were developing interfaces to go through that process of definition, before even mentioning the process of testing. I couldn't conceive of how we were going to impart that sociological change and manage the cost associated with doing that.

BS Let's follow that last thought—that this was as much a sociological as a technical concern—with a more general one: this whole question of evolving a more rigorous engineering of the interface. Tell us a bit more about how you got the engineers to come along on this trip.

DB Step one is recognizing the key problem. Then a couple of long-haired guys—maybe as I was once upon a time—think about how to introduce some new technology or rocket science to go after this. This is necessary, but immensely far from sufficient to solve the problem.

What are the follow-on steps—after you've come up with these beautiful brainchildren and maybe implemented some glossy prototype that shows people how to do it? That is the rest of the "journey of 1,000 miles." Whether you can actually get all of the engineers, the *n*-hundred engineers who might put a new interface into a library in Solaris, to engage in these practices is a fundamentally social question. You have to succeed at this to have your foundation for stability. It's a contract, and you have to maintain it; otherwise, it has no value.

This is a pretty big hurdle because you're adding overhead to what an engineer does. You're putting another rock in his or her backpack. You're saying, "Not only must it be both 32- and 64-bit capable, both SPARC and x86, work properly under multithreading, be internationalized, etc."—the whole list of requirements that any project must already meet—"but now you also have to define all these interfaces carefully and make sure that they're subject to proper upward compatibility."

That is a social change management problem. Part of the reason for picking these highly simplistic necessity-focused as opposed to sufficiency-focused kinds of interface definitional practices is that there's some vague hope that you can actually get the engineers to take this on board, and that you can build the tools to enforce this—that is to say, observe that it has been done in all cases and audit the interface from release to release to ensure that it has maintained upward compatibility.

BS I think you could put that slightly more positively. This is the hallmark of good engineering: You didn't try to boil the ocean. You didn't try to set an impossible goal or one that only certain gurus could practice. You found a middle ground that made huge improvements over the alternatives, yet that everybody could practice. Innovation is still distributed throughout the organization. Anybody is perfectly capable of going through the process of adding functionality to the system. The tests and the tools are all out there for everybody to use. You preserved the essential social structure of the engineering enterprise.

DB Yes, I was mostly being emphatic about respecting what other people in the organization do, and realizing how hard their job is and how much leverage you need to give them in order to get them to do it.

The first hurdle is your own engineers, but then what makes this a really huge task is that you've also got to get all of your third-party application developers and middleware developers on board with this idea. And you've got to get your end-user customers on board to explain to them what the value proposition is that you're trying to impart to them, so that they'll be the force feedback that pulls on everybody else in the ecosystem to deliver these benefits (in particular the ISVs who are sometimes a little bit at odds with this from a business perspective).

For me, this is a big awakening. Starting out as an engineer and being a technical software systems guy made it easy to underestimate that the vast majority of the effort is in communicating succinctly what the problem is, what the benefit is going to be, and then what everybody is going to have to do to retread the ecosystem to get to this better place.

Then there is constant reiteration and support, demonstrating that we're not just wagging our lips, we're really doing this. There's this persistence, not just over one or two releases, that we have this bright idea. We started this back in Solaris 2.4, and here we are 12 years and seven releases later in Solaris 10, and these practices are in place. They're institutional in our organization.

BS What do you think are the major lessons that you've learned over the years? Obviously, keep it simple, set achievable goals, look for the main effect. But maybe you could impart other lessons about how systems are structured or built or evolve in the first place.

DB A really important colleague in my life was Paul Haeberli, whom I worked with at Silicon Graphics in the early days, and he once said to me, "You know Dave, it's not about these big genius ideas. What really makes a great system is thousands of detailed engineering decisions that to a first approximation we got all pretty much right."

There's definitely an element of that in here, which is very hard to talk usefully about. It's this conceptual shift, in this case a zealousness that you have to have about compatibility and engineering for compatibility, and the belief that we're going to go to reasonably extreme measures to come up with specific techniques. People such as Tim Marsland, who's now a Sun Fellow, were responsible for a lot of these things in the implementation when we did the binary compatibility between SunOS 4 and SunOS 5—all the retrofitting of SunOS 5 to build bridges so that those earlier SunOS 4 binaries would continue to run without change in SunOS 5 (i.e., Solaris 2).

BS Or to put it slightly differently, buried beneath what may look like a simple result is some pretty tricky engineering.

DB Sometimes you get some high-level realizations and you can base a whole campaign on it as I did with the ABI. I said, "Let's just focus on necessity: What are the things that every application has to do, and how can we just take an initial whack at this?" That's what we did with this library and symbol-level of definition of the interface. It's very simple and very far from sufficient to ensure robustness. But I would point out that a lot of the robustness comes from a whole bunch of other things that happen, whether it's the architectural review process where the senior engineers look at everything, or all the testing we do before we let a release out.

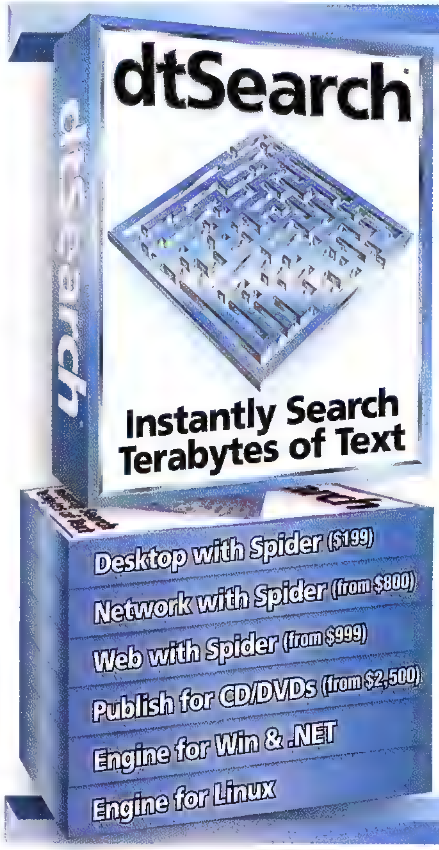
There's just a lot that goes on that's extremely important and not necessarily glorified. There are a lot of "quiet heroes," as Marsland once said, who have done incredibly important and unsung things to make this a reality.

Another high-level lesson for me is that certain attitudes and points of view are important in the way you come at this, things such as respecting other people in the organization and the amount of work you might be giving them and really looking for the value proposition and who's going to get it, and where the incentive feedback is coming from.

At an even higher level, I think that some developers don't even address this because they think innovation and stability are mutually exclusive. But it turns out that you have to change your perspective and realize you can impart new—arguably revolutionary—functionality, but in an evolutionary sort of way. This is the mindset you need to have. Q

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums


dtSearch
Instantly Search Terabytes of Text
Desktop with Spider ($199)
Network with Spider (from $800)
Web with Spider (from $999)
Publish for CD/DVDs (from $2,500)
Engine for Win & .NET
Engine for Linux

dtSearch® Terabyte Indexer

"Bottom line: dtSearch manages a terabyte of text in a single index and returns results in less than a second" — *InfoWorld*

- over two dozen indexed, unindexed, fielded data and full-text search options
- highlights hits in HTML, XML and PDF, while displaying links, formatting and images
- converts other file types (word processor, database, spreadsheet, email and attachments, ZIP, Unicode, etc.) to HTML for display with highlighted hits
- Spider supports static and dynamic Web content, with WYSWYG hit-highlighting
- optional API for C++, .NET, Java, SQL, etc. Ask about new .NET Spider API

Developer Quotes and Reviews

dtSearch vs. the competition: **"dtSearch easily overpowered the document indexing and searching abilities of other solutions, especially against large volumes of documents"**

Reliability: **"dtSearch got the highest marks from our systems engineering folks that I've ever heard of"**

Results: **"customer response has been phenomenal"**

For hundreds more reviews and developer case studies, see www.dtsearch.com

Contact dtSearch for fully-functional evaluations

"For combing through large amounts of data, dtSearch ... leads the market" — *Network Computing*

"Blindingly fast" — *Computer Forensics: Incident Response Essentials*

"Super fast, super-reliable" — *The Wall Street Journal*

"A powerful arsenal of search tools" — *The New York Times*

"Powerful Web-based engines" — *eWeek*

"Blazing speeds" — *Computer Reseller News Test Center*

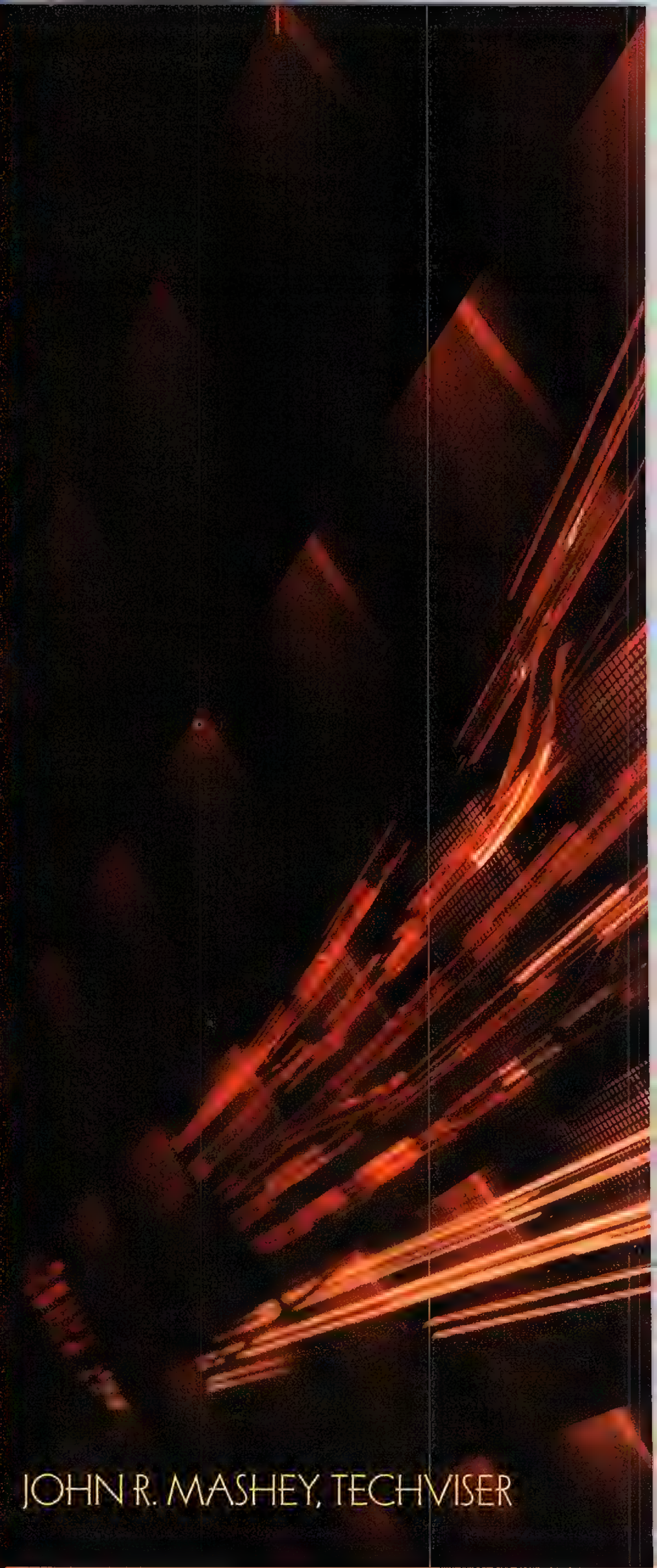
"The most powerful document search tool on the market" — *Wired Magazine*

The Smart Choice for Text Retrieval® since 1991

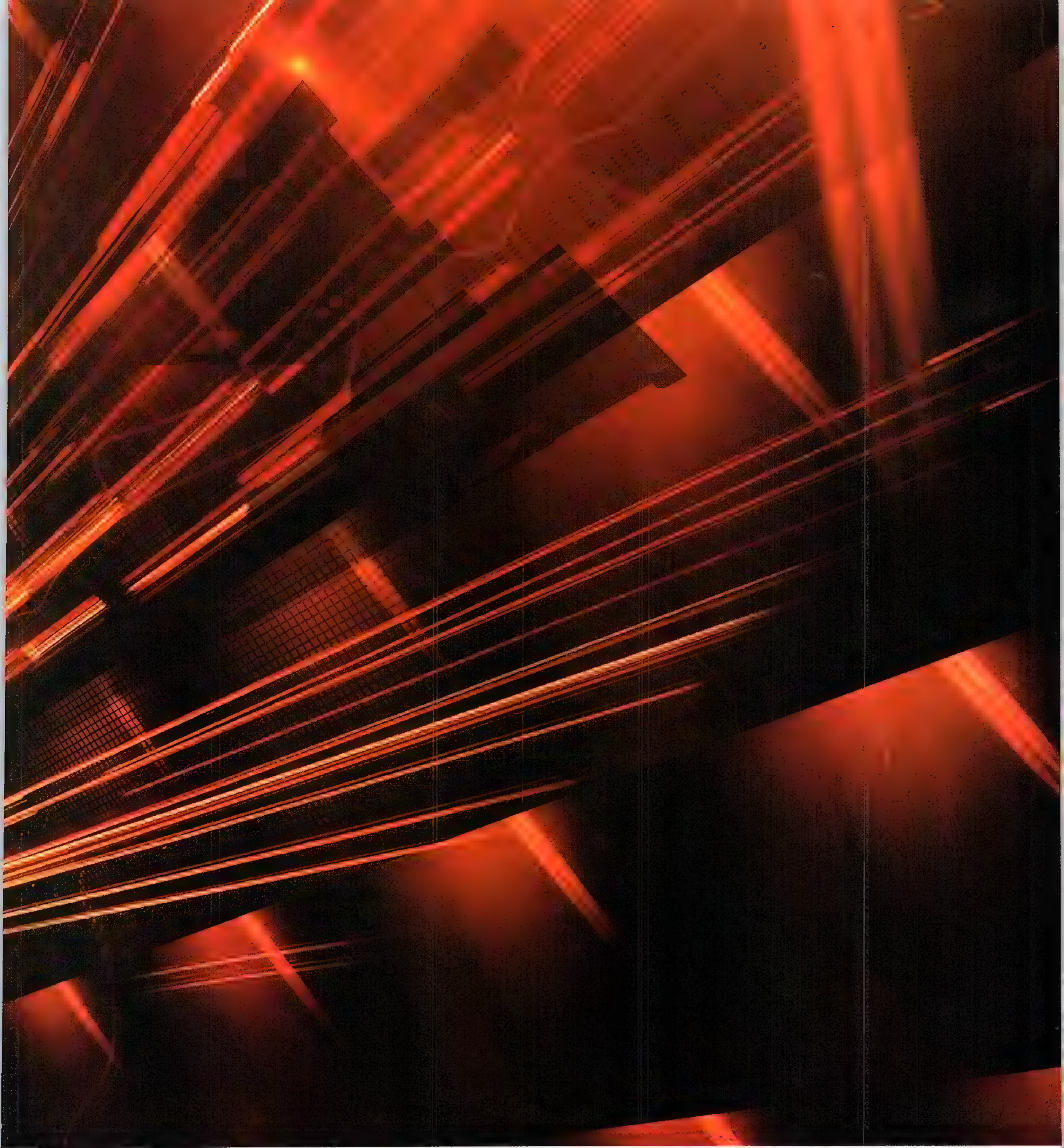
1-800-IT-FINDS • www.dtsearch.com

THE LONG ROAD TO 64 BITS

JOHN R. MASHEY, TECHVISER



Double, double, toil and trouble—Shakespeare, *Macbeth*



Shakespeare's words (*Macbeth*, Act 4, Scene 1) often cover circumstances beyond his wildest dreams. *Toil and trouble* accompany major computing transitions, even when people plan ahead. To calibrate "tomorrow's legacy today," we should study "tomorrow's legacy yesterday." Much of tomorrow's software will still be driven by decades-old decisions. Past decisions have unanticipated side effects that last decades and can be difficult to undo.

For example, consider the overly long, often awkward, and sometimes contentious process by which 32-bit microprocessor systems evolved into 64/32-bitters needed to address larger storage and run mixtures of 32- and 64-bit user programs. Most major general-purpose CPUs now have such versions, so bits have "doubled," but "toil and trouble" are not over, especially in software.

This example illustrates the interactions of hardware,

THE LONG ROAD TO 64 BITS

languages (especially C), operating system, applications, standards, installed-base inertia, and industry politics. We can draw lessons ranging from high-level strategies down to programming specifics.

FUNDAMENTAL PROBLEM (LATE 1980S)

Running out of address space is a long tradition in computing, and often quite predictable. Moore's law increased the size of DRAM approximately four times every three to four years, and by the mid-1990s, people were able to afford 2 to 4 GB of memory for midrange microprocessor systems, at which point simple 32-bit addressing (4 GB) would get awkward. Ideally, 64/32-bit CPUs would have started shipping early enough (1992) to have made up the majority of the relevant installed base before they were actually needed. Then people could have switched to 64/32-bit operating systems and stopped upgrading 32-bit-only systems, allowing a smooth transition. Vendors naturally varied in their timing, but shipments ranged from "just barely in time" to "rather late." This is somewhat odd, considering the long, well-known histories of insufficient address bits, combined with the clear predictability of Moore's law. All too often, customers were unable to use memory they could easily afford.

Some design decisions are easy to change, but others create long-term legacies. Among those illustrated here are the following:

- Some unfortunate decisions may be driven by real constraints (1970: PDP-11 16-bit).
- Reasonable-at-the-time decisions turn out in 20-year retrospect to have been suboptimal (1976-77: usage of C data types). Some better usage recommendations could have saved a great deal of toil and trouble later.
- Some decisions yield short-term benefits but incur long-term problems (1964: S/360 24-bit addresses).
- Predictable trends are ignored, or transition efforts underestimated (1990s: 32 transitioning to 64/32 bits).

Constraints. Hardware people needed to build 64/32-bit CPUs at the right time—neither too early (extra cost, no market), nor too late (competition, angry customers). Existing 32-bit binaries needed to run on upward-compatible 64/32-bit systems, and they could be expected to coexist forever, because many would never need to be 64 bits. Hence, 32 bits could not be a temporary compatibility feature to be quickly discarded in later chips.

Software designers needed to agree on whole sets of standards; build dual-mode operating systems, compilers, and libraries; and modify application source code to work in both 32- and 64-bit environments. Numerous details had to be handled correctly to avoid redundant hardware efforts and maintain software sanity.

Solutions. Although not without subtle problems, the hardware was generally straightforward and not that expensive—the first commercial 64-bit micro's 64-bit data path added at most 5 percent to the chip area, and this fraction dropped rapidly in later chips. Most chips used the same general approach of widening 32-bit registers to 64 bits. Software solutions were much more complex, involving arguments about 64/32-bit C, the nature of existing software, competition/cooperation among vendors, official standards, and influential but totally unofficial ad hoc groups.

Legacies. The IBM S/360 is 40 years old and still supports a 24-bit legacy addressing mode. The 64/32 solutions are at most 15 years old, but will be with us, effectively, forever. In 5,000 years, will some software maintainer still be muttering, "Why were they so dumb?"[1]

We managed to survive the Y2K problem—with a lot of work. We're still working through 64/32. Do we have any other problems like that? Are 64-bit CPUs enough to help the "Unix 2038" problem, or do we need to be working harder on that? Will we run out of 64-bit systems, and what will we do then? Will IPv6 be implemented widely enough soon enough?

All of these are examples of long-lived problems for which modest foresight may save later toil and trouble. But software is like politics: Sometimes we wait until a problem is really painful before we fix it.

PROBLEM: CPU MUST ADDRESS AVAILABLE MEMORY

Any CPU can efficiently address some amount of virtual memory, done most conveniently by flat addressing in which all or most of the bits in an integer register form a virtual memory address that may be more or less than actual physical memory. Whenever affordable physical memory exceeds the easily addressable, it stops being easy to throw memory at performance problems, and programming complexity rises quickly. Sometimes, segmented memory schemes have been used with varying degrees of success and programming pain. History is filled with awkward extensions that added a few bits to extend product life a few years, usually at the cost of hard work by operating-system people.

Moore's law has increased affordable memory for decades. Disks have grown even more rapidly, especially since 1990. Larger disk pointers are more convenient than smaller ones, although less crucial than memory pointers. These interact when mapped files are used, rapidly consuming virtual address space.

In the mid-1980s, some people started thinking about 64-bit micros—for example, the experimental systems built by DEC (Digital Equipment Corporation). MIPS Computer Systems decided by late 1988 that its next design must be a true 64-bit CPU, and it announced the R4000 in 1991. Many people thought MIPS was crazy or at least premature. I thought the system came just barely in time to develop software to match increasing DRAM, and I wrote an article to explain why.[2] The issues have not changed very much since then.

N-bit CPU. By long custom, an N-bit CPU implements an ISA (instruction set architecture) with N-bit integer registers and N (or nearly N) address bits, ignoring sizes of buses or floating-point registers. Many "32-bit" ISAs have 64- or 80-bit floating-point registers and implementations with 8-, 16-, 32-, 64-, or 128-bit buses. Sometimes marketers have gotten this confused. I use the term *64/32-bit* here to differentiate the newer microprocessors from the older 64-bit word-oriented supercomputers, as the software issues are somewhat different. In the same sense, the Intel 80386 might have been called a 32/16-bit CPU, as it retained complete support for the earlier 16-bit model.

Why 2N-bits? People sometimes want wider-word computers to improve performance for parallel bit operations or data movement. If you need a 2N-bit operation (add, multiply, etc.), each can be done in one instruction on a 2N-bit CPU, but longer sequences are required on an N-bit CPU. These are straightforward low-level performance issues. The typical compelling reason for wider words, however, has been the need to increase address bits, because code that is straightforward and efficient with enough address bits may need global restructuring to survive fewer bits.

Addressing—virtual and real—in a general-purpose system. User virtual addresses are mapped to real memory addresses, possibly with intervening page faults whereby the operating system maps the needed code or data from disk into memory. A user program can access at most VL (virtual limit) bytes, where VL starts at some hardware limit, then sometimes loses more space to an operating system. For example, 32-bit systems easily have VLs of 4, 3.75, 3.5, or 2 GB. A given program execution uses at most PM (program memory) bytes of virtual memory. For many programs PM can differ greatly according to the input, but of course $PM \le VL$.

The RL (real limit) is visible to the operating system and is usually limited by the width of physical address buses. Sometimes mapping hardware is used to extend RL beyond a too-small "natural" limit (as happened in PDP-11s, described later). Installed AM (actual memory) is less visible to user programs and varies among machines without needing different versions of the user program.

Most commonly, $VL \ge RL \ge AM$. Some programs burn virtual address space for convenience and actually perform acceptably when $PM >> AM$: I've seen cases where 4:1 still worked, as a result of good locality. File mapping

THE LONG ROAD TO 64 BITS

can increase that ratio further and still work. On the other hand, some programs run poorly whenever PM > AM, confirming the old proverb, "Virtual memory is a way to sell real memory."

Sometimes, a computer family starts with VL ≥ RL ≥ AM, and then AM grows, and perhaps RL is increased in ways visible only to the operating system, at which point VL << AM. A single program simply cannot use easily buyable memory, forcing work to be split and making it more difficult. For example, in Fortran, the declaration REAL X(M, M, M) is a three-dimensional array. If M=100, X needs 4 MB, but people would like the same code to run for M=1,000 (4 GB), or 6,300 (1,000 GB). A few such systems do exist, although they are not cheap. I once had a customer complain about lack of current support for 1,000 GB of real memory, although later the customer was able to buy such a system and use that memory in one program. After that, the customer complained about lack of 10,000-GB support....

Of course, increasing AM in a multitasking system is still useful in improving the number of memory-resident tasks or reducing paging overhead, even if each task is still limited by VL. Operating system code is always simplified if it can directly and simply address all installed memory without having to manage extra memory maps, bank registers, etc.

Running out of address bits has a long history.

MAINFRAMES, MINICOMPUTERS, MICROPROCESSORS

The oft-quoted George Santayana is appropriate here: "Those who cannot remember the past are condemned to repeat it."

Mainframes. IBM S/360 mainframes (circa 1964; see accompanying Chronology sidebar) had 32-bit registers, of which only 24 bits were used in addressing, for a limit of 16 MB of core memory. This was considered immense at the time. A "large" mainframe offered at most 1 MB of memory, although a few "huge" mainframes could provide 6 MB. Most S/360s did not support virtual memory, so user programs generated physical addresses directly, and the installed AM was partitioned among the operating system and user programs. The 16-MB limit was unfortunate, but ignoring (not trapping) the high-order 8 bits was worse. Assembly language programmers "cleverly" packed 8-bit flags with 24-bit addresses into 32-bit words.

As virtual addressing S/370s (1970) enabled programs that were larger than physical memory allowed, and as core gave way to DRAM (1971), the 16-MB limit grew inadequate. IBM 370-XA CPUs (1983) added 31-bit addressing mode, but retained a (necessary) 24-bit mode for upward compatibility. I had been one of those "clever" programmers and was somewhat surprised to discover that a large program (the S/360 ASSIST assembler) originally written in 1970 was still in use in 2006—in 24-bit compatibility mode, because it wouldn't run any other way. Compiler code that had been "clever" had long since stopped doing this, but assembly code is tougher. ("The evil that men do lives after them, the good is oft interred with their bones." —Shakespeare, again, *Julius Caesar*)

Then, even 31-bit addressing became insufficient for certain applications, especially databases. ESA/370 (1988) offered user-level segmentation to access multiple 2-GB regions of memory.

The 64-bit IBM zSeries (2001) still supports 24-bit mode, 40-plus years later. Why did 24-bit happen? I'm told that it was all for the sake of one low-cost early model, the 360/30, where 32 bits would have run slower because it had 8-bit data paths. These were last shipped more than 30 years ago. Were they worth the decades of extra headaches?

Minicomputers. In the 16-bit DEC PDP-11 minicomputer family (1970), a single task addressed only 64 KB, or in later models (1973), 64 KB of instructions plus 64 KB of data. "The biggest and most common mistake that can be made in computer design is that of not providing enough address bits for memory addressing and management," C. Gordon Bell and J. Craig Mudge wrote in 1978. "The PDP-11 followed this hallowed tradition of skimping on address bits, but was saved on the principle that a good design can evolve through at least one major change. For the PDP-11, the limited address space was solved for the short run, but not with enough finesse to support a large family of minicomputers. This was indeed a costly oversight."[3]

To be fair, it would have been difficult to meet the PDP-11's cost goals with 32-bit hardware, but I think DEC did underestimate how fast the price of DRAM memory

Chronology: Multiple Interlocking Threads

1964	**IBM S/360**: 32-bit, with 24-bit addressing (16 MB total) of real (core) memory.
1968	**Algol 68**: includes long long.
1970	**DEC PDP-11/20**: 16-bit, 16-bit addressing (64 KB total).
	IBM S/370 family: virtual memory, 24-bit addresses, but multiple user address spaces allowed.
1971	**IBM 370/145**: main memory no longer core, but DRAM, 1 Kbit/chip.
1973	**DEC PDP-11/45**: separate instruction+data (64 KI + 64 KD); 248 KB maximum real memory.
	Unix: PDP-11/45, operating system rewritten in C; IP16.
	C: integer data types: int, char; C on other machines (36-bit Honeywell 6000, IBM 370, others).
1975	**Unix**: sixth edition, 24-bit maximum file size (16 MB).
1976	**DEC PDP-11/70**: (64 KI + 64 KD), but larger physical memory (a huge 4 MB).
	C: short, long added (partly from doing C for XDS Sigma, although long was 64 bits there).
1977	**Unix**: ported to 32-bit Interdata 8/32.
	C: unsigned, typedef, union; 32-bit long used to replace int[2] in lseek, tell on 16-bit PDP-11; IP16L32.
	DEC VAX-11/780: 32-bit, 32-bit addressing (4 GB total, 2 GB per user process).
	C: **PDP-11**: I16LP32; **VAX** (other 32-bitters): ILP32.
1978	**Unix**: 32V for VAX-11/780; C is ILP32.
	C: *The C Programming Language*, Brian Kernighan and Dennis Ritchie (Prentice-Hall).
	Intel 8086: 16-bit, but with user-visible segmentation.
1979	**Motorola MC68000**: 32-bit ISA, but 24-bit addressing (e.g., S/360).
1982	**C**: I16LP32 on MC68000 in Bell Labs Blit terminal.
	Intel 80286: allows 16 MB of real memory, but restrictions keep most systems at 1 MB.
1983	**IBM 370/XA**: adds 31-bit mode for user programs; 24-bit mode still supported.
	C: Unix workstations generally use ILP32, following Unix on VAX systems.
1984	**Motorola MC68020**: 32-bit; 32- bit addressing.
	C: Amdahl UTS (32-bit S/370) uses long long, especially for large file pointers.
	C: Convex (64-bit vector mini-supercomputer) uses long long for 64-bit integers.
1986	**Intel**: 80386, 32-bit, with support for 8086 mode.
1987	**Apple Mac II**: MC68020's 32-bit addressing causes trouble for some MC68000 software.
1988	**IBM ESA/370**: multiple 31-bit address spaces per user, although complex; 24-bit still there.
1989	**ANSI C** ("C89"): effort had started in 1983, ANSI X3J11.
1992	**SGI**: ships first 64-bit micro (MIPS R4000); still running 32-bit operating system.
	64-bit C working group: discusses various models (LP64, ILP64, LLP64), with little agreement.
	DEC: ships 64-bit Alpha systems, running 64-bit operating system; LP64.
1994	**SGI**: ships IRIX 6 (64/32 operating system; ILP32LL + LP64) on Power Challenge; customers buy 4 GB+ memory, use it.
	DEC: ships 4 GB+ in DEC 7000 SMPs (may have been slightly earlier).
1995	**Sun UltraSPARC**: 64/32-bit hardware, 32-bit-only operating system.
	HAL Computer's SPARC64: uses ILP64 model for C.
	Large file summit: codifies 64-bit interface to files >2 GB, even in 32-bit systems (ILP32LL+LP64).
	Aspen group: supports LP64 model for C so that Unix vendors are consistent.
1996	**HP**: announces PA-RISC 2.0, 64-bit.
1997	**HP**: UP/UX 11.0 is 64/32-bit OS; ILP32LL + LP64.
	IBM: RS64 PowerPC, AIX 4.3; ILP32LL + LP64.
1998	**Sun**: 64/32 Solaris 7 released; ILP32LL + LP64.
1999	**C**: ISO/IEC C (WG14's "C99"); includes long long, at least 64 bits.
2001	**IBM**: 64-bit zSeries (S/360 descendant); 24-bit addressing still supported.
	Intel: 64-bit Itanium.
2002	**Microsoft**: Windows 64-bit for Itanium.
2003	**AMD**: 64-bit X86 (now called AMD64).
2004	**Intel**: 64-bit X86 (called EMT64), compatible with AMD.
2005	**Microsoft**: Windows XP Professional x64 for X86; LLP64 (or IL32LLP64).

THE LONG ROAD TO 64 BITS

would fall. In any case, this lasted a long time—the PDP-11 was finally discontinued in 1997!

The PDP-11/70 (1976) raised the number of supportable concurrent tasks, but any single program could still use only 64 KI (instruction) + 64 KD (data) of a maximum of 4 MB, so that individual large programs required unnatural acts to split code and data into 64 KB pieces. Some believed this encouraged modularity and inhibited "creeping featurism" and was therefore philosophically good.

Although the 32-bit VAX-11/780 (1977) was only moderately faster than the PDP-11/70, the increased address space was a major improvement that ended the evolution of high-end PDP-11s. VAX architect William Strecker explained it this way: "However, there are some applications whose programming is impractical in a 65-KB address space, and perhaps more importantly, others whose programming is appreciably simplified by having a large address space."[4]

Microprocessors. The Intel 8086's 16-bit ISA seemed likely to fall prey to the PDP-11's issues (1978). It did provide user-mode mechanisms for explicit segment manipulation, however. This allowed a single program to access more than 64 KB of data. PC programmers were familiar with the multiplicity of memory models, libraries, compiler flags, extenders, and other artifacts once needed. The 80386 provided 32-bit flat addresses (1986), but of course retained the earlier mechanisms, and 16-bit PC software lasted "forever." The intermediate 80286 (1982) illustrated the difficulty of patching an architecture to get more addressing bits.

The 32-bit Motorola MC68000 (1979) started with a flat-addressing programming model. By ignoring the high 8 bits of a 32-bit register, it exactly repeated the S/360 mistake. Once again, "clever" programmers found uses for those bits, and when the MC68020 (1984) interpreted all 32, some programs broke—for example, when moving from the original Apple Macintosh to the Mac II (1987).

Fortunately, 64-bit CPUs managed to avoid repeating the S/360 and MC68000 problem. Although early versions usually implemented 40 to 44 virtual address bits, they trapped use of not-yet-implemented high-order v bits, rather than ignoring them. People do learn, eventually.

LESSONS

- Even in successful computer families created by top architects, address bits are scarce and are totally consumed sooner or later.
- Upward compatibility is a real constraint, and thinking ahead helps. In the mainframe case, a 24-bit "first-implementation artifact" needed hardware/software support for 40-plus years. Then a successful minicomputer family's evolution ended prematurely. Finally, microprocessors repeated the earlier mistakes, although the X86's segmentation allowed survival long enough to get flat-address versions.

THE 32- TO 64-BIT PROBLEM IN THE LATE 1980S

By the late '80s, Moore's law seemed cast in silicon, and it was clear that by 1993-94, midrange microprocessor servers could cost-effectively offer 2-4 GB or more of physical memory. We had seen real programs effectively use as much as 4:1 more virtual memory than installed physical memory, which meant pressure in 1993-94, and real trouble by 1995. As I wrote in *BYTE* in September 1991:[5]

"The virtual addressing scheme often can exceed the limits of possible physical addresses. A 64-bit address can handle literally a mountain of memory: Assuming that 1 megabyte of RAM requires 1 cubic inch of space (using 4-megabit DRAM chips), 2**64 bytes would require a square mile of DRAM piled more than 300 feet high! For now, no one expects to address this much DRAM, even with next-generation 16-MB DRAM chips, but increasing physical memory slightly beyond 32 bits is definitely a goal. With 16-MB DRAM chips, 2**32 bytes fits into just over 1 cubic foot (not including cooling)—feasible for deskside systems....

"Database systems often spread a single file across several disks. Current SCSI disks hold up to 2 gigabytes (i.e., they use 31-bit addresses). Calculating file locations as virtual memory addresses requires integer arithmetic. Operating systems are accustomed to working around such problems, but it becomes unpleasant to make workarounds; rather than just making things work well, programmers are struggling just to make something work.... "

So, people started to do something about the problem.

SGI (Silicon Graphics). Starting in early 1992, all new SGI products used only 64/32-bit chips, but at first they still ran a 32-bit operating system. In late 1994, a 64/32-bit operating system and compilers were introduced for large servers, able to support both 32-bit and 64-bit user programs. This software worked its way down the product line. A few customers quickly bought more than 4 GB of memory and within a day had recompiled programs to use it, in some cases merely by changing one Fortran parameter. Low-end SGI workstations, however, continued to ship with a 32-bit-only operating system for years, and of course, existing 32-bit hardware had to be

For any successful computer family it takes a very long time to convert an installed base of hardware, and software lasts even longer.

supported—for years. For historical reasons, SGI had more flavors of 32-bit and 64-bit instruction sets than were really desirable, so it was actually worse than having just two of them.

This is the bad kind of "long tail"—people focus on "first ship dates," but often the "last ship date" matters more, as does the "last date on which we will release a new version of an operating system or application that can run on that system." Windows 16-bit applications still run on regular Windows XP 20 years after the 80386 was introduced. Such support has finally been dropped in Windows XP x64.

DEC. DEC shipped 64-bit Alpha systems in late 1992, with a 64-bit operating system, and by late 1994 was shipping servers with memories large enough to need greater than 32-bit addressing. DEC might have requested (easy) 32-bit ports, but thinking long term, it went straight to 64-bit, avoiding duplication. It was expensive in time and money to get third-party software 64-bit clean, but it was valuable to the industry as it accelerated the 64-bit clean-up. DEC was probably right to do this, since it had no installed base of 32-bit Alpha programs and could avoid having to support two modes. For VMS, early versions were 32-bit, and later ones 64/32-bit.

Other vendors. Over the next few years, many vendors shipped 64-bit CPUs, usually running 32-bit software, and later 64/32-bit: Sun UltraSPARC (1995), HAL SPARC64 (1995), PA-RISC (1996), HP/UX 11.0 (1997), IBM RS64 and AIX 4.3 (1997), Sun Solaris 7 (1998), IBM zSeries (2001), Intel Itanium (2001), AMD AMD64 (2003), Intel EMT64a (2004), Microsoft Windows XP x64 (2005). Linux 64-bit versions appeared at various times.

Most 64-bit CPUs were designed as extensions of existing 32-bit architectures that could run existing 32-bit binaries well, usually by extending 32-bit registers to 64 bits in 64-bit mode, but ignoring the extra bits in 32-bit mode. The long time span for these releases arises from natural differences in priorities. SGI was especially interested in high-performance technical computing, whose users were accustomed to 64-bit supercomputers and could often use 64 bits simply by increasing one array dimension in a Fortran program and recompiling. SGI and other vendors of large servers also cared about memory for large database applications. It was certainly less important to X86 CPU vendors whose volume was dominated by PCs. In Intel's case, perhaps the emphasis on Itanium delayed 64-bit X86s.

In 2006, 4 GB of DRAM, consisting of 1-GB DRAMs, typically uses four DIMMs and can cost less than $400. At least some large notebooks support 4 GB, and 300-GB disks are widely available for less than $1 per gigabyte, so one would expect mature, widespread support for 64 bits by now. All this took longer than perhaps it should have, however, and there have been many years where people could buy memory but not be able to address it conveniently, or not be able to buy some third-party application that did, because such applications naturally lag 64-bit CPUs. It is worth understanding why and how this happened, even though the impending issue was surely well known.

LESSONS

- For any successful computer family, it takes a very long time to convert an installed base of hardware, and software lasts even longer.
- Moving from 32 to 64/32 bits is a long-term coexistence scenario. Unlike past transitions, almost all 64-bit CPUs must run compatible existing 32-bit binaries without

THE LONG ROAD TO 64 BITS

translation, since much of the installed base remains 32-bit, and many 32-bit programs are perfectly adequate and can remain so indefinitely.

SOFTWARE IS HARDER: OPERATING SYSTEM, COMPILERS, APPLICATIONS, USERS, STANDARDS

Building a 64-bit CPU is not enough. Embedded-systems markets can move more easily than general-purpose markets, as happened, for example, with Cisco routers and Nintendo N64s that used 64-bit MIPS chips. Vendors of most 32-bit systems, however, had to make their way through all of the following steps to produce useful upward-compatible 64-bit systems:

1. Ship systems with 64/32-bit CPUs, probably running in 32-bit mode. Continue supporting 32-bit-only CPUs as long as they are shipped and for years thereafter (often five or more years). Most vendors did this, simply because software takes time.

2. Choose a 64-bit programming model for C, C++, and other languages. This involves discussion with standards bodies and consultation with competitors. There may be serious consequences if you select a different model from most of your competitors. Unix vendors and Microsoft did choose differently, for plausible reasons. Think hard about inter-language issues—Fortran expects INTEGER and REAL to be the same size, which makes 64-bit default integers awkward.

3. Clean up header files, carefully.

4. Build compilers to generate 64-bit code. The compilers themselves almost certainly run in 32-bit mode and cross-compile to 64-bit, although occasional huge machine-generated programs can demand compilers that run in 64-bit mode. Note that programmer sanity normally requires a bootstrap step here, in which the 32-bit compiler is first modified to accept 64-bit integers and then is recoded to use them itself.

5. Convert the operating system to 64-bit, but with 32-bit interfaces as well, to run both 64- and 32-bit applications.

6. Create 64-bit versions of all system libraries.

7. Ship the new operating system and compilers on new 64-bit hardware, and hopefully, on the earlier 64-bit hardware that has now been shipping for a while. This includes supporting (at least) two flavors of every library.

8. Talk third-party software vendors into supporting a 64-bit version of any program for which this is relevant. Early in such a process, the installed base inevitably remains mostly 32-bit, and software vendors consider the potential market size versus the cost of supporting two versions on the same platform. DEC helped the industry fix 32- to 64-bit portability issues by paying for numerous 64-bit Alpha ports.

9. Stop shipping 32-bit systems (but continue to support them for many years).

10. Stop supporting 32-bit hardware with new operating system releases, finally.

Going from step 1 to step 6 typically took two to three years, and getting to step 9 took several more years. The industry has not yet completed step 10.

Operating system vendors can avoid step 1, but otherwise, the issues are similar. Many programs need never be converted to 64-bit, especially since many operating systems already support 64-bit file pointers for 32-bit programs.

The next section traces some of the twists and turns that occurred in the 1990s, especially involving the implementation of C on 64/32-bit CPUs. This topic generated endless and sometimes vituperative discussions. Interested readers should consult references 6-9. Masochistic readers might search newsgroup comp.std.c for "64-bit" or "long long".

C: 64-BIT INTEGERS ON 64/32-BIT CPUS—TECHNOLOGY AND POLITICS

People have used various (and not always consistent) notations to describe choices of C data types. In table 1, the first label of several was the most common, as far as I can tell. On machines with 8-bit char, short is usually 16 bits, but other data types can vary. The common choices are shown in table 1.

Early days. Early C integers (1973) included only int and char; then long and short were added by 1976, followed by unsigned and typedef in 1977. In the late 1970s, the installed base of 16-bit PDP-11s was joined by newer 32-bit systems, requiring that source code be efficient, sharable, and compatible between 16-bit systems (using I16LP32) and 32-bitters (ILP32), a pairing that worked well. PDP-11s still employed (efficient) 16-bit int most of the time, but could use 32-bit long as needed. The 32-bitters used 32-bit int most of the time, which was more efficient, but could express 16-bit via short. Data structures used to communicate among machines avoided int. It was very important that 32-bit long be usable on the PDP-11. Before that, the PDP-11 needed explicit functions to manipulate int[2] (16-bit int pairs), and such code was not only awkward, but also not simply sharable with 32-bit systems. This is an extremely important point—long was not strictly necessary for 32-bit CPUs, but it was very important to enable code sharing among 16- and 32-bit environments. We could have gotten by with char, short, and int, if all our systems had been 32 bits.

It is important to remember the nature of C at this point. It took a while for typedef to become a common idiom. With 20/20 hindsight, it might have been wise to have provided a standard set of typedefs to express "fast integer," "guaranteed to be exactly N-bit integer," "integer large enough to hold a pointer," etc., and to have recommended that people build their own typedefs on these definitions, rather than base types. If this had been done, perhaps much toil and trouble could have been avoided.

This would have been very countercultural, however, and it would have required astonishing precognition. Bell Labs already ran C on 36-bit CPUs and was working hard on portability, so overly specific constructs such as "int16" would have been viewed with disfavor. C compilers still had to run on 64 KI + 64 KD PDP-11s, so language minimality was prized. The C/Unix community was relatively small (600 systems) and was just starting to adapt to the coming 32-bit minicomputers. In late 1977, the largest known Unix installation had seven PDP-11s, with a grand total of 3.3 MB of memory and 1.9 GB of disk space. No one could have guessed how pervasive C and its offshoots would become, and thinking about 64-bit CPUs was not high on the list of issues.

32-bit happy times. In the 1980s, ILP32 became the norm, at least in Unix-based systems. These were happy times: 32-bit was comfortable enough for buyable DRAM for many years. In retrospect, however, it may have caused some people to get sloppy in assuming: sizeof(int) == sizeof(long) == sizeof(ptr) == 32.

Sometime around 1984, Amdahl UTS and Convex added long long for 64-bit integers, the former on a 32-bit architecture, the latter on a 64-bitter. UTS used this especially for long file pointers, one of the same motivations for long in PDP-11 Unix (1977). Algol 68 inspired long long in 1968, and it was also added to GNU C at some point. Many reviled this syntax, but at least it consumed no more reserved keywords.

Of course, 16-bit int was used on Microsoft DOS and Apple Macintosh systems, given the original use of Intel 8086 or MC68000, where 32-bit int would have been costly, particularly on early systems with 8- or 16-bit data paths and where low memory cost was especially important.

64-bit heats up in 1991/1992. The MIPS R4000 and DEC Alpha were announced in the early 1990s. E-mail discussions were rampant among various companies during 1990-92 regarding the proper model for 64-bit C, especially when implemented on systems that would still run 32-bit applications for many years. Quite often, such

TABLE 1 Common C Data Types

int	long	ptr	long long	Label	Examples
16	--	16	--	IP16	PDP-11 Unix (early, 1973)
16	32	16	--	IP16L32	PDP-11 Unix (later, 1977); multiple instructions for long
16	32	32	--	I16LP32	Early MC68000 (1982); Apple Macintosh 68K Microsoft operating systems (plus extras for X86 segments)
32	32	32	--	ILP32	IBM 370; VAX Unix; many workstations
32	32	32	64	ILP32LL or ILP32LL64	Amdahl; Convex; 1990s Unix systems Like IP16L32, for same reason; multiple instructions for long long.
32	32	64	64	LLP64 or IL32LLP64 or P64	Microsoft Win64
32	64	64	*(64)	LP64 or I32LP64	Most 64/32 Unix systems
64	64	64	*(64)	ILP64	HAL; logical analog of ILP32

**In these cases, LP64 and ILP64 offer 64-bit integers, and long long seems redundant, but in practice, most proponents of LP64 and ILP64 included long long as well, for reasons given later. ILP64 uniquely required a new 32-bit type, usually called _int32.*

THE LONG ROAD TO 64 BITS

informal cooperation exists among engineers working for otherwise fierce competitors.

In mid-1992 Steve Chessin of Sun initiated an informal 64-bit C working group to see if vendors could avoid implementing randomly different 64-bit C data models and nomenclatures. Systems and operating system vendors all feared the wrath of customers and third-party software vendors otherwise. DEC had chosen LP64 and was already far along, as Alpha had no 32-bit version. SGI was shipping 64-bit hardware and working on 64-bit compilers and operating system; it preferred LP64 as well. Many others were planning 64-bit CPUs or operating systems and doing portability studies.

Chessin's working group had no formal status, but had well-respected senior technologists from many systems and operating system vendors, including several who were members of the C Standards Committee. With all this brainpower, one might hope that one clear answer would emerge, but that was not to be. Each of the three proposals (LLP64, LP64, and ILP64) broke different kinds of code, based on the particular implicit assumptions made in the 32-bit happy times.

Respected members of the group made credible presentations citing massive analyses of code and porting experience, each concluding, "XXX is the answer." Unfortunately, XXX was different in each case, and the group remained split three ways. At that point I suggested we perhaps could agree on header files that would help programmers survive (leading to <inttypes.h>). Most people did agree that long long was the least bad of the alternative notations and had some previous usage.

We worried that we were years ahead of the forthcoming C standard but could not wait for it, and the C Standards Committee members were supportive. If we agreed on anything reasonable, and it became widespread practice, it would at least receive due consideration. Like it or not, at that point this unofficial group probably made long long inevitable—or perhaps that inevitability dated from 1984.

By 1994, DEC was shipping large systems and had paid for many third-party software ports, using LP64. SGI was also shipping large systems, which supported both ILP32LL and LP64, with long long filling the role handled by long in the late 1970s.

The DEC effort proved that it was feasible to make much software 64-bit-clean without making it 32-bit unclean. The SGI effort proved that 32-bit and 64-bit programs could be sensibly supported on one system, with reasonable data interchange, a requirement for most other vendors. In practice, that meant that one should avoid long in structures used to interchange data, exactly akin to the avoidance of int in the PDP-11/VAX days. About this time in 1995 the Large File Summit agreed on Unix APIs to increase file size above 2 GB, using long long as a base data type.

Finally, the Aspen Group in 1995 had another round of discussions about the 64-bit C model for Unix and settled on LP64, at least in part because it had been proved to work and most actual 64-bit software used LP64.

During the 1992 meetings of the 64-bit C group, Microsoft had not yet chosen its model, but later chose LLP64, not the LP64 preferred by Unix vendors. I was told that this happened because the only 32-bit integer type in PCs was long; hence, people often used long to mean 32-bit more explicitly than in Unix code. That meant that changing its size would tend to break more code than in Unix. This seemed plausible. Every choice broke some code; thus, people looked at their own code bases, which differed, leading to reasonable differences of opinion.

Many people despised long long and filled newsgroups with arguments against it, even after it became incorporated into the next C standard in 1999. The official rationale for the C standard can be consulted by anyone who wants to understand the gory details.[6]

Differing implicit assumptions about the sizes of various data types had grown up over the years and caused a great deal of confusion. If we could go back to 1977, knowing what we know now, we could have made all this easier simply by insisting on more consistent use of typedef. In 1977, however, it would have been hard to think ahead 20 years to 64-bit micros—we were just getting to 32-bit minis!

This process might also have been eased if more vendors had been further along in 64-bit implementations in 1992. Many people had either forgotten the lessons of the PDP-11/VAX era or had not been involved then. In any case, the 64/32 systems had a more stringent requirement: 64-bit and 32-bit programs would coexist forever in the same systems.

LESSONS

• Standards often get created in nonstandard ways, and often, de facto standards long precede official ones.
• Reasonable people can disagree, especially when looking at different sets of data.
• Sometimes one has to work with competitors to make anything reasonable happen.
• Programmers take advantage of extra bits or ambiguity of specification. Most arguments happen because application programmers make differing implicit assumptions.
• Code can be recompiled, but once data gets written somewhere, any new code must still be able to describe it cleanly. Current software is rarely done from scratch but has to exist inside a large ecosystem.
• We might never build 128-bit computers, but it would probably be good to invent a notation for 128-bit integers, whose generated code on 64-bit CPUs is about the same as 64-bit code is on 32-bit CPUs. It would be nice to do that long before it is really needed. In general, predictable long-term problems are most efficiently solved with a little planning, not with frenzied efforts when the problem is imminent. Fortunately, 128-bitters are many years away, if ever (maybe 2020-2040), because we've just multiplied our addressing size by 4 billion, and that will last a while, even if Moore's law continues that long! In case 128-bit happens in 2020, however, it would be wise to be thinking about the next integer size around 2010.

Of course, when the S/360 was introduced, IBM and other vendors had 36- and 18-bit product lines. In an alternate universe, had the S/360 been a 36-bit architecture with four 9-bit bytes/word, most later machines would have been 18- and 36-bit, and we would just be starting the 36-bit to 72-bit transition.
• Hardware decisions last a long time, but software decisions may well last longer. If you are a practicing programmer, take pity on those who end up maintaining your code, and spend some time thinking ahead. Allow for natural expansion of hardware, hide low-level details, and use the highest-level languages you can. C's ability to make efficient use of hardware can be both a blessing and a curse.

CONCLUSIONS

Some decisions last a very long time. The 24-bit addressing of 1964's S/360 is still with us, as are some side effects of C usage in the mid-1970s. The transition to 64-bit probably took longer than it needed for a host of reasons. It's too bad that people quite often have been unable to use affordable memory for solving performance problems or avoiding cumbersome programming.

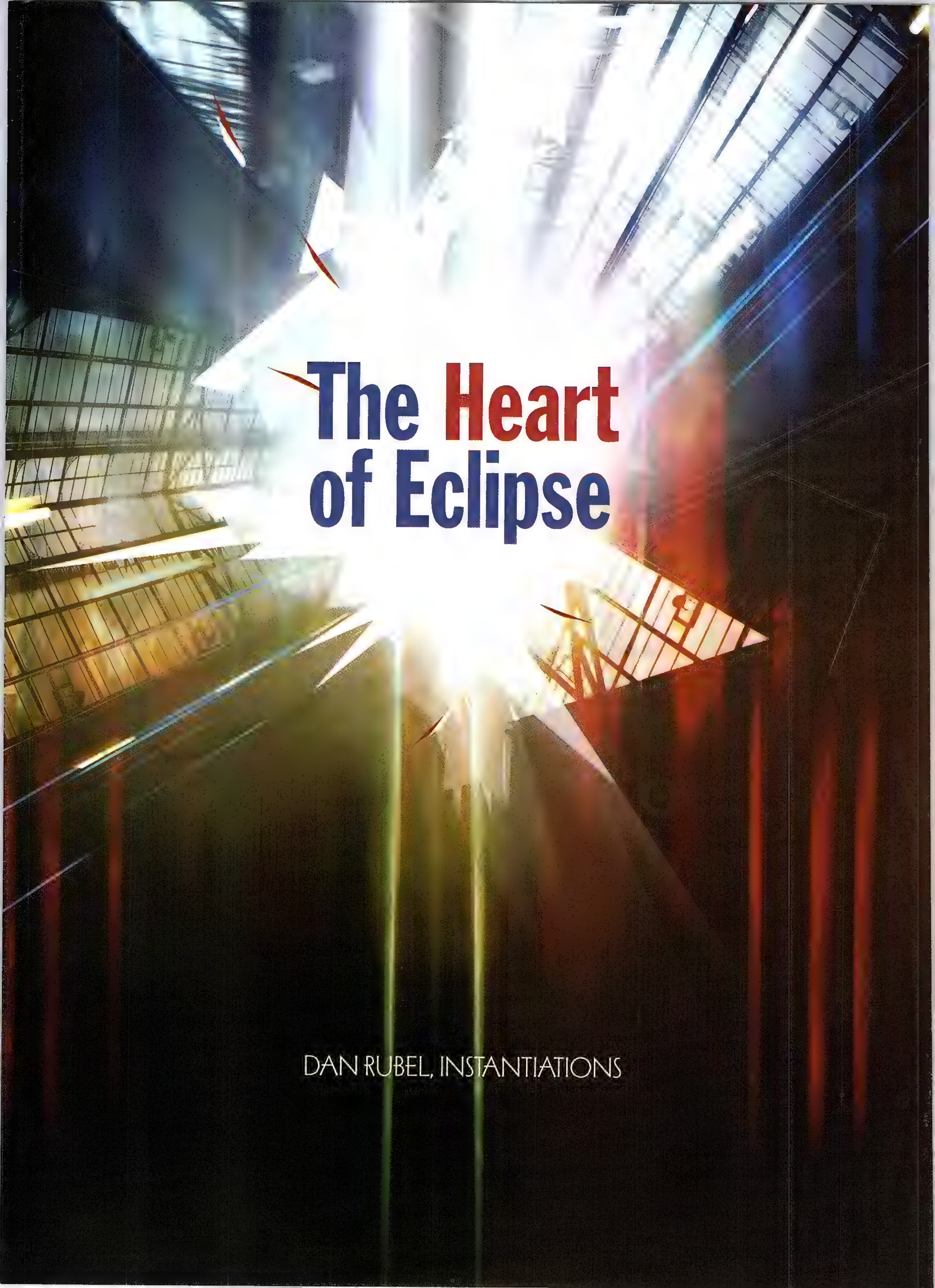
It's too bad there has been so much toil and trouble, but at least people have stopped arguing about whether there should be 64-bit micros or not. Q

REFERENCES

1. Mashey, J. 2004-05. Languages, levels, libraries, longevity. *ACM Queue* 2 (9): 32-38.
2. Mashey, J. 1991. 64-bit computing. *BYTE* (September): 135-142. The complete text can be found by searching Google Groups comp.arch: Mashey BYTE 1991.
3. Bell, C. G., Mudge, J. C. 1978. The evolution of the PDP-11. In *Computer Engineering: A DEC View of Computer System Design*, ed. C. Gordon Bell, J. Craig Mudge, and John E. McNamara. Bedford, MA: Digital Press.
4. Strecker, W. D. 1978. VAX-11/780: A virtual address extension to the DEC PDP-11 family. In *Computer Engineering: A DEC View of Computer System Design*, ed. C. Gordon Bell, J. Craig Mudge, and John E. McNamara. Bedford, MA: Digital Press.
5. See reference 2.
6. Rationale for International Standard—Programming Languages—C; http://www.open-std.org/jtc1/sc22/wg14/www/docs/n897.pdf (or other sites).
7. Aspen Data Model Committee. 1997-1998. 64-bit programming models: Why LP64? http://www.unix.org/version2/whatsnew/lp64_wp.html.
8. Josey, A. 1997. Data size neutrality and 64-bit support; http://www.usenix.org/publications/login/standards/10.data.html.
9. Adding support for arbitrary file sizes to the single Unix specification; http://www.unix.org/version2/whatsnew/lfs20mar.html.

LOVE IT, HATE IT? LET US KNOW
feedback@acmqueue.com or www.acmqueue.com/forums

JOHN MASHEY is a consultant for venture capitalists and technology companies, sits on various technology advisory boards of startup companies, and is a trustee of the Computer History Museum. He spent 10 years at Bell Laboratories, working on the Programmer's Workbench version of Unix. He later managed development of Unix-based applications and software development systems. After Bell Labs, he was one of the designers of the MIPS RISC architecture, one of the founders of the SPEC benchmarking group, and chief scientist at Silicon Graphics. His interests have long included interactions between hardware and software. He received a B.S. in mathematics, and M.S. and Ph.D. in computer science from Pennsylvania State University.



The Heart of Eclipse

DAN RUBEL, INSTANTIATIONS

A look inside an extensible plug-in architecture

ECLIPSE is both an open, extensible development environment for building software and an open, extensible application framework upon which software can be built. Considered the most popular Java IDE, it provides a common UI model for working with tools and promotes rapid development of modular features based on a plug-in component model (figure 1). The Eclipse Foundation (http://www.eclipse.org) designed the platform to run natively on multiple operating systems, including Macintosh, Windows, and Linux, providing robust integration with each and providing rich clients that support the GUI interactions everyone is familiar with: drag and drop, cut and paste (clipboard), navigation, and customization. You can think of Eclipse as a "design center" supported by a development team of 300 or more developers whom you can leverage when developing your own software.

Although designed as a universal tool-integration platform, Eclipse is not only for creating IDEs. It is just as helpful for constructing general-purpose applications—for example, workflow, help systems, or contact management systems (figure 2). In fact, NASA used Eclipse RCP (Rich Client Platform) to build Maestro, a software program for managing remote vehicles on space missions.

ARCHITECTURAL OVERVIEW OF ECLIPSE RCP

At the heart of Eclipse is an architecture for dynamically discovering, loading, and running components, or *plug-ins*, the basic unit of functionality in Eclipse. The plug-ins determine the platform functionality and whether it operates as an IDE or a general-purpose application.

The Eclipse IDE is separated into layers, each consisting of smaller-grained plug-ins. The Java IDE is layered on top of a more general-purpose tools platform, which in turn is layered on the Eclipse RCP framework (figure 2). Eclipse RCP provides plug-ins that form a very basic application infrastructure, including services such as the windowing system, the help system, preference pages, and an update manager (see table 1).

The development objectives for Eclipse and Eclipse RCP are to:

- Foster cross-platform development for Windows, Macintosh, Linux, and other operating systems
- Support native-user interface elements ("widgets") rather than emulate them on supported operating systems
- Reduce the memory footprint and startup time of components and plug-ins
- Create reusable modules for general application development

The Eclipse teams strive to reduce the effort of designing an application by providing a sort of "Chinese menu" of readily available modules (plug-ins), thus decreasing development time and time-to-market, while still building world-class applications. Eclipse plug-ins are loosely coupled, and developers are free to snap them together in different configurations using only those necessary to solve a particular problem.

Plug-ins for RCP can be programmed to the Eclipse portable APIs and run unchanged on any supported operating system. This allows developers to focus on designing plug-ins that perform important business tasks, such as workflow, processing, diagramming, reporting, publishing, or any other business requirement. There are many plug-ins generally available for Eclipse, or users can design their own. The Eclipse platform handles the logistics of finding and running the right code, and the platform UI provides a standard model for navigation. (For a list of plug-ins available, go to http://www.eclipse-plugincentral.com.)

Eclipse RCP contains five fundamental parts:

- UI Workbench with editors, perspectives, and views.
- SWT (Standard Widget Toolkit), a low-level graphics library of Java GUI components with native implementations that hide differences between platforms or operating systems so that the developer can program to a single API for all platforms.
- JFace, a GUI abstraction layer for displaying objects that is layered on top of SWT.
- Platform Runtime, which defines the extension-point model facilitating loose coupling between plug-ins and just-in-time lazy loading and initialization.
- OSGi (Open Services Gateway Initiative), a framework used by Eclipse for plug-in discovery and lifecycle management, including loading and unloading plug-ins without requiring the application to be restarted.

ECLIPSE UI WORKBENCH

The Eclipse UI Workbench organizes the user experience and supplies the structures in which an application interacts with the user. From the user's standpoint, an application window consists visually of *editors*, *views*, and *actions* that can be rearranged by the user to better suit the task. *Perspectives* manifest themselves in the selection and arrangements of editors, views, and actions visible on the screen (figure 3). The UI Workbench frees developers from having to reinvent a high-performance user interface for each Java-client application running on different operating systems.

Overview of the Eclipse Development Environment

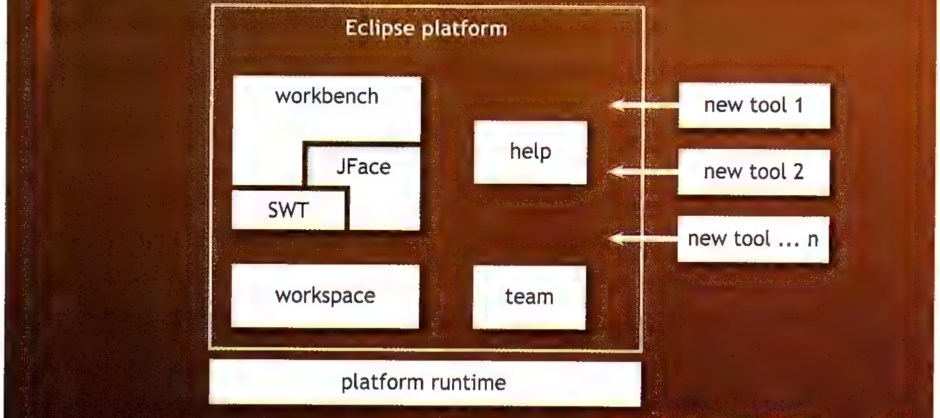


FIG 1

Editors allow the user to open, edit, and save domain objects, such as a record of a person's name, age, contact information, and social security number. Editors follow an open-save-close lifecycle much as file-system-based tools do, but they are more tightly integrated into the workbench. When active, an editor can contribute actions to the workbench menus and tool bar.

Views provide information about objects related to the user's current task, such as a list of objects that can be edited. A view may assist an editor by providing information about the document being edited. For example, if the editor implements a standard interface, then the content outline view shows a structured outline for the content of the active editor. A view may augment other views by providing information about a selected object. Views have simpler lifecycles than editors. Modifications made in a view (such as changing a property value) are saved immediately, and the changes are reflected immediately in other related parts of the UI.

Actions permit user commands to be defined independently from their exact location in the UI. An action represents a command that can be triggered by the user with a button, menu item, or tool-bar item. Each action knows its own UI properties (label, icon, tool tip, etc.) used to construct appropriate widgets for presenting the action. This separation allows the same action to be used in several places in the UI and means that it is easy to change where an action is presented in the UI without

Overview of a Client Application Built on Eclipse RCP

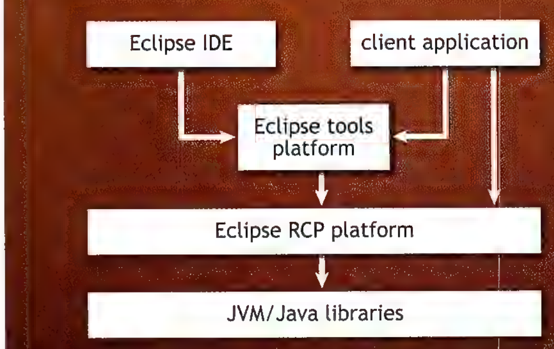


FIG 2

TABLE 1 Details of Eclipse

Platform Runtime	Dynamically discovers plug-ins and maintains information about the plug-ins and their services in a platform registry. Plug-ins are loaded and launched when required, according to user operation of the platform. The runtime is implemented using the OSGi framework.
Resource management (workspace)	Defines API for creating and managing resources (projects, files, and folders) that are produced by tools and kept in the file system.
Eclipse UI Workbench	Implements the user cockpit for navigating the platform. It defines extension points for adding UI components such as views or menu actions. It supplies additional toolkits (JFace and SWT) for building user interfaces. The UI services are structured so that a subset of the UI plug-ins can be used to build rich client applications that are independent of the resource management and workspace model. IDE-centric plug-ins define additional function for navigating and manipulating resources.
Help system	Defines extension points for plug-ins to provide help or other documentation as browsable books.
Team support	Defines a team programming model for managing and versioning resources.
Debug support	Defines a language-independent debug model and UI classes for building debuggers and launchers.
Other utilities	Other utility plug-ins supply functions such as searching and comparing resources, performing builds using XML configuration files, and dynamically updating the platform from a server.

having to change the code for the action itself.

Perspectives organize views and editors in an arrangement suitable for a specific user task. Views and editors can be tiled, stacked, or detached for presentation on the screen. A perspective controls initial view visibility, layout, and action visibility, but once the perspective has been opened, the user has complete control and can easily rearrange and customize a perspective to better suit a particular task. At any time, the user can quickly switch to a different perspective to work on a different task. Multiple perspectives can be opened in a single window or in separate windows, all controlled by the user.

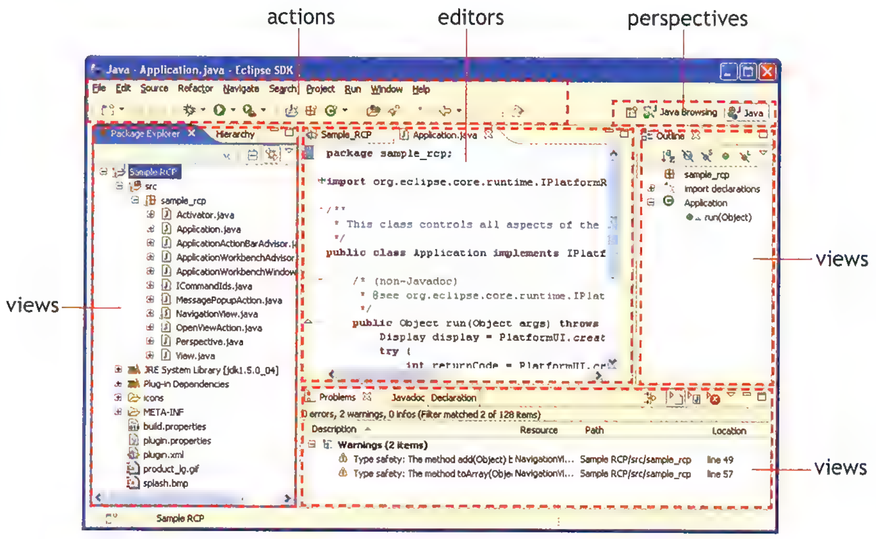
The Eclipse workbench window offers a set of visual design choices about how a UI is organized. It has a menu bar, tool bar, shortcut bar, and a status line, along with certain colors and gradients that give things an integrated and distinctive appearance. Views have titles with actions in their title bar; the workbench has File, Edit, Window, and Help listings. Part of every perspective is set aside as an editor area. For a general-purpose application, each of these UI elements can be visible or suppressed, based upon the needs of the application.

The user's ability to configure and customize is an important part of the UI. For example, the workbench allows the user to rearrange elements in the perspective by dragging and dropping views and maximizing a view by double-clicking on its title bar. A general-purpose application might want to decide whether and how the user can affect the layout. A user might want to rearrange how a person's information appears on the screen—for example, by making it smaller to see more people's information or even rearranging the order of the information.

New functionality integrates into the UI's editors-views-perspectives paradigm in well-defined ways. All views and editors implement common interfaces and access common APIs so that users may customize their appearance, content, and location in the application window. Extension points allow plug-ins to augment the workbench by adding new types of editors, views, actions, and perspectives.

The workbench API is implemented using both SWT and JFace for a platform native user experience. AWT (Abstract Window Toolkit) and Swing are not used because AWT provides too small a set of native widgets, and Swing provides emulated and not

Eclipse Workbench Screenshot



```
Sample RCP
  src
    sample_rcp
      Activator.java
      Application.java
      ApplicationActionBarAdvisor.j
      ApplicationWorkbenchAdvisor
      ApplicationWorkbenchWindow
      ICommandIds.java
      MessagePopupAction.java
      NavigationView.java
      OpenViewAction.java
      Perspective.java
      View.java
  JRE System Library [jdk1.5.0_04]
  Plug-in Dependencies
  icons
  META-INF
  build.properties
  plugin.properties
  plugin.xml
  product_lg.gif
  splash.bmp
```

```
package sample_rcp;

+import org.eclipse.core.runtime.IPlatformR

/**
 * This class controls all aspects of the
 */
public class Application implements IPlatf

    /* (non-Javadoc)
     * @see org.eclipse.core.runtime.IPlat
     */
    public Object run(Object args) throws
        Display display = PlatformUI.creat
        try {
            int returnCode = PlatformUI.cr
```

```
sample_rcp
import declarations
Application
  run(Object)
```

```
0 errors, 2 warnings, 0 infos (Filter matched 2 of 128 items)
Description                                              Resource       Path                          Location
Warnings (2 items)
  Type safety: The method add(Object) b NavigationVi...  Sample RCP/src/sample_rcp   line 49
  Type safety: The method toArray(Obje NavigationVi...   Sample RCP/src/sample_rcp   line 57
```

FIG 3

native widgets; furthermore, at the time both lacked some of the performance needed. Today client applications can use AWT and Swing alongside SWT and JFace, although typically only SWT and JFace are used.

SWT

SWT provides a common, operating system-independent API for widgets and graphics. The entire Eclipse platform UI and the tools that plug into it use SWT for presenting information to the user. SWT is implemented to allow tight integration with the underlying native window system. It provides an infrastructure for modular, native widgets consistent with the platform the application targets. This results in a responsive, high-quality user experience.

A low-level graphics library of Java GUI components with native implementations, SWT hides differences between platforms or operating systems so that a developer can program to a single API for all platforms using a consistent set of high-performance widgets. This means developers can write the code once and make it look like a Macintosh or Windows application UI. As part of the UI Workbench, SWT also contributes to giving the user a responsive, high-quality experience with the Java-client application that feels native to the platform. For small, lightweight applications to a low-level API where a tiny footprint is important, such as a mobile application, developers might choose SWT over JFace.

The tension between portable toolkits and native window system integration is always an issue with widget toolkit design. Java AWT provides low-level widgets (lists, text fields, and buttons), but no high-level widgets (trees or rich text). AWT widgets are implemented directly with native widgets on all underlying window systems. Building a UI using AWT alone means programming to the least common denominator of all operating system window systems.

The Java Swing toolkit addresses this issue by emulating widgets such as trees, tables, and rich text. Swing also provides look-and-feel emulation layers that attempt to make applications look like the underlying native window system. Emulated widgets invariably lack the look-and-feel of native widgets, however. Users interacting with emulated widgets notice enough difference from shrink-wrapped software applications and their native widgets that it is hard for software with an emulated UI to compete.

To overcome this problem, SWT defines a common API available across a number of supported window systems. Wherever possible, SWT uses native widgets for each different native window system. When a native widget is available on one platform but not another, SWT provides a suitable emulation. SWT implements common low-level widgets such as lists, text fields, and buttons everywhere using native widgets; some generally useful higher-level widgets, however, may need to be emulated on some window systems. This approach allows SWT to maintain a consistent programming model across all environments and lets the look-and-feel of the underlying native window system show through as much as possible.

Internally, the SWT implementation provides separate and distinct implementations in Java for each native window system while keeping the API identical across all platforms. The Java native libraries are completely different for each platform, with each tying the specific underlying window system to the common API with a minimum amount of code. Tight integration with the underlying native window system is not strictly a matter of look-and-feel. SWT also interacts with native desktop features such as drag-and-drop, allowing the user to drag-and-drop content from SWT widgets to other native applications not based on SWT.

In cases where a particular underlying native window system provides a unique and significant feature that cannot be emulated on other window systems, SWT exposes the native window system-specific APIs. Windows ActiveX is an example. The window system-specific API is segregated into appropriately named packages to indicate that it is inherently nonportable. Because these window system-specific APIs are implemented with a minimum amount of code, the SWT implementation is expressed entirely in Java code that looks familiar to the native operating system developer. Windows programmers using C will find the Java implementation of SWT for Windows ActiveX instantly familiar, since it consists of calls to the Windows API that they already know.

This strategy greatly simplifies implementing, debugging, and maintaining SWT because it allows all interesting development to be done in Java. On the other hand, this is not a concern for ordinary SWT clients because native widgets are hidden behind the window system-independent SWT API.

JFACE

JFace is a Java UI toolkit with classes for handling many common UI programming tasks at a higher level. SWT and JFace are separate because they solve different problems: SWT provides a common API across all platforms, whereas JFace builds upon this common yet simple API to provide an easy-to-use UI abstraction layer. Using JFace, programmers interact with their own domain objects rather than their more primitive underlying UI presentation. For example, to display a list of records, you implement an interface describing what should be displayed for a given record, and from that point on you manipulate lists of records rather than lists of strings representing those records.

JFace is window-system-independent in both its API and implementation and includes the usual UI toolkit components of image and font registries, dialog, preference, wizard frameworks, and progress reporting for long-running operations. Three of its more interesting features are *viewers*, *content providers*, and *label providers*.

Viewers provide object-oriented wrappers around their associated SWT components. They provide higher-level semantics than SWT widgets. The standard viewers for lists, trees, and tables support populating the viewer with elements from the client's domain and keeping the widgets in sync with changes to that domain. These viewers are configured with a content provider and a label provider and can optionally be configured with element-based filters and sorters.

Content providers know how to map the viewer's input element to expected viewer content and how to translate domain changes into appropriate viewer updates.

Label providers produce the specific string label and icon needed to display any given domain element in the widget.

Clients are notified of selections and events in terms of the domain elements they provide to the viewer. The viewer implementation handles the mapping between domain elements and SWT widgets, adjusting for a filtered view of the elements and re-sorting when necessary. The standard viewer for text supports common operations such as double-click behavior, undo, coloring, or navigating by character index or line number. Text viewers provide a document model to the client and manage the conversion of the document to the information required by the SWT-styled text widget.

Existing domain objects can be displayed in the Eclipse UI by implementing the appropriate interfaces. Multiple viewers can be open on the same model or document; all are updated automatically when the model or document changes in any of them. If you are show-

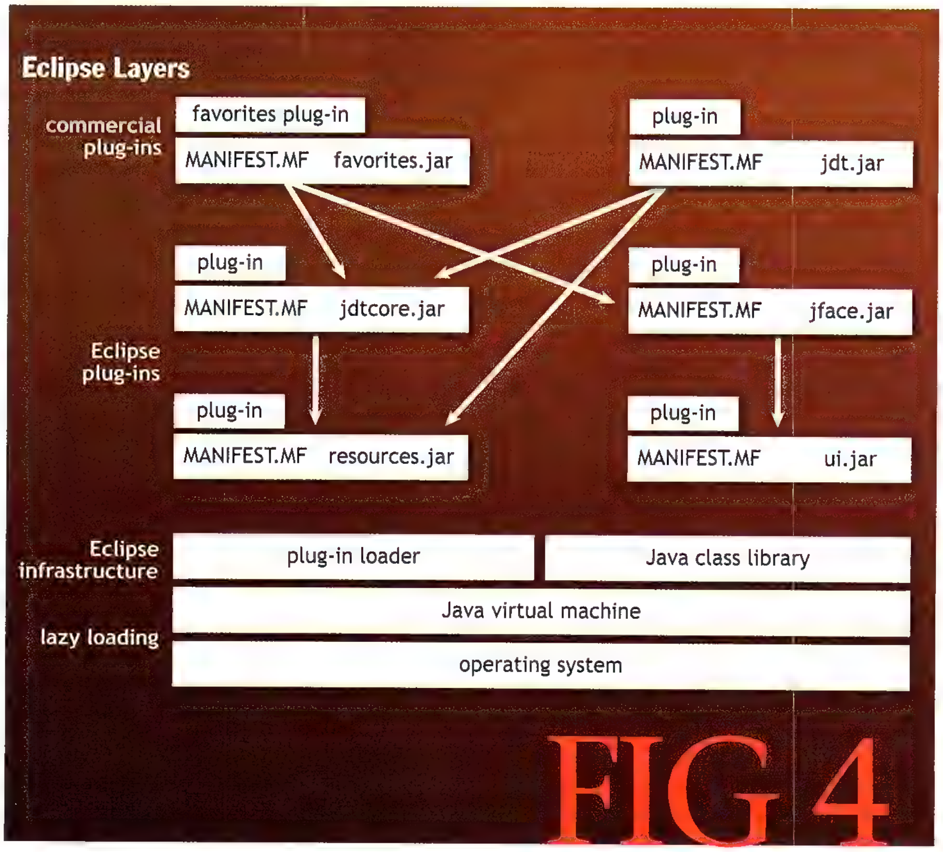


FIG 4

ing a list of record information, for example, and the user selects a particular element in the list, then internal notifications are generated informing other user interface elements so that they can update their content based on the new selection.

PLATFORM RUNTIME

The Platform Runtime facilitates loosely coupled program modules (plug-ins) through the use of extensions and extension points. Plug-ins declare how they extend other plug-ins and how other plug-ins can extend them. Using this information, the Platform Runtime dynamically determines the minimum set of plug-ins necessary for a given task and loads only those plug-ins; this reduces the memory footprint of the application and improves the startup speed.

The Platform Runtime uses the OSGi framework to determine which plug-ins are available when an application is running. This information, along with the plug-in extension and extension-point information, provides each plug-in with information about which plug-ins extend it and how. Because this information is not hard-coded into each plug-in, but instead dynamically discovered at time of execution, programmers can more easily recombine plug-ins in different ways to solve different problems without having to reinvent the wheel.

OSGI FRAMEWORK

The OSGi framework is a thin layer that allows multiple Java-based components to cooperate in a single JVM (Java Virtual Machine). It focuses on the plug-in lifecycle so that plug-ins can be installed, updated, or removed on the fly without disrupting the operation of the device. Its software components are libraries or applications that can dynamically discover and use other components (see figure 4).

Components are lazily loaded on an as-needed basis, reducing the startup time and overall memory footprint of the application. Many standard component interfaces are available for OSGi, including common functions such as HTTP servers, configuration, logging, security, user administration, and XML. Plug-in-compatible implementations of these components can be obtained from different vendors with different optimizations. OSGi adds the following functionalities to the Eclipse RCP platform:

- A powerful and rigidly specified Java-based class-loading layer defining the class-loading policies.
- A lifecycle layer that adds bundles that can be dynamically installed, started, stopped, updated, and uninstalled. Bundles rely on the module layer for class loading but add an API to manage the modules in runtime.
- A service registry providing a cooperation model for bundles that facilitates service discovery without loading and executing the bundle.

PLUG-INS AND THEIR OPERATION

A plug-in, also known as a bundle, is the smallest functional unit of the Eclipse platform that can be developed and delivered separately. Plug-ins are coded in Java. Very small applications might be written as a single plug-in, whereas complex applications blend their functionality across several plug-ins. Except for the Platform Runtime, all of the Eclipse platform's functionality is located in plug-ins. They can provide support for editing and manipulating additional types of resources such as Java files, C programs, Word documents, HTML pages, and JSP (JavaServer pages) files. Plug-ins determine the ultimate functionality of the platform or application. That's why the Eclipse SDK ships with additional plug-ins to enhance its functionality.

A typical plug-in contains a manifest file, Java code in a JAR (Java archive) library, some read-only files, and other resources such as images, Web templates, message catalogs, native-code libraries, etc. A plug-in, however, may not contain code at all. A plug-in that contributes online help in the form of HTML pages is one example. The plug-in code libraries and read-only content are located together in a directory in the file system or at a base URL on a server. Another mechanism permits a plug-in to be synthesized from separate fragments, each in its own directory or URL. This is the mechanism used to deliver separate language packs for an internationalized plug-in.

The manifest file of a plug-in defines how that plug-in interacts with the system. This includes information uniquely identifying the plug-in and version. It also includes what services that plug-in consumes and what services that plug-in provides to other plug-ins. This declarative approach to modularization allows the Eclipse Platform Runtime to load only the plug-ins necessary for the current operation of the application.

CHALLENGES OF BUILDING APPLICATIONS USING ECLIPSE RCP

Developers wanting to use Eclipse RCP face several technical issues, including getting up to speed and writing loosely coupled code:

Getting up to speed. Eclipse is a very well-designed and well-supported object-oriented framework. It represents a very large API and therefore can have a steep learning curve for someone new to the environment. Many books, training materials, and online articles are available to provide a general understanding and direct developers where to dig for more details. How to use a particular function or feature is well documented in code via the Javadoc explaining the API.

Stick to the API. Many developers are used to writing code that accesses internal code in other modules. Taking the same approach for an Eclipse RCP application means the code will not be maintainable, because the internal code changes as the Eclipse platform evolves.

Loosely coupled code. Many developers are used to writing code interacting with other modules but not writing a manifest declaring *how* their code interacts with other modules. Writing plug-ins that declare how they interact with other plug-ins promotes flexibility and reusability of the code being written and is a cornerstone of Eclipse-based development. Developers need to think in terms of what "services" a plug-in provides and how it extends or consumes "services" of other plug-ins.

Testing tools. The availability of testing tools can also be an issue. For example, few automated GUI testing tools are available for Eclipse RCP applications; JUnit is a standard Java test framework but does not easily address UI testing without additional functionality layered on top of it. Fortunately, some third-party solutions are emerging.

Installers. Eclipse RCP doesn't have an installer, making deployment and application updates more difficult. Commercial third-party installers, however, can handle this well.

Plug-in quality, maturity, and support. For some, Eclipse is rich enough as an application framework that it doesn't require other plug-ins to add capabilities. Many organizations, however, need to enhance Eclipse RCP with features and functions that match their business requirements. This means they will have to dip into the Eclipse plug-in ecosystem, which brings up several issues. A parallel can be found in the commercial software world where freeware and shareware are offered. The quality, maturity, and support of these products are obvious considerations.

Solutions are emerging that address identification, quality, and support issues. SourceLabs validates, integrates, and supports open source products. Eclipse Plug-in Central (http://www.eclipseplugincentral.com) is a clearinghouse for Eclipse plug-ins. Yoxos is another example of Eclipse distribution that includes open source plug-ins available in managed scenarios and supported.

COMMERCIAL THIRD-PARTY TOOLS

For developers choosing to create a rich client for their applications, there are commercial tools, such as RCP Developer (http://www.instantiations.com/rcpdeveloper), available. These tools help with not only the GUI design and coding, but also UI testing, creating help documentation, and packaging and deployment of the application. The update manager in Eclipse can keep users up to date with the latest versions of their applications.

CONCLUSION

Many organizations are standardizing on RCP as the architecture for their internal IT applications, thus enabling them to run as a manageable, integrated suite rather than as disassociated units of functionality. Eclipse RCP will have serious implications for the desktop computing strategies of enterprise organizations because it incorporates application development tools that are easy to use, provides an open source extensible application framework, and creates applications that can run on a variety of platforms. Q

LOVE IT, HATE IT? LET US KNOW
feedback@acmqueue.com or www.acmqueue.com/forums

DAN RUBEL is the CTO at Instantiations. An expert in the design and application of object-oriented technologies, he has more than 15 years of software development experience, including eight years of experience with Java and four years with Eclipse. He is the primary architect and product manager for several commercial products. He has a B.S. from Bucknell and is co-author of the book *Eclipse: Building Commercial Quality Plug-ins* (Addison-Wesley Eclipse series).

Sponsored by

USENIX & THE USENIX SIG FOR [sage] SYSADMINS

6 DAYS OF TRAINING
BY INDUSTRY EXPERTS, INCLUDING:

- Gerald Carter on Ethereal and the Art of Debugging Networks
- Richard Bejtlich on TCP/IP Weapons
- Æleen Frisch on Administering Linux in Production Environments
- Chip Salzenberg on Higher-Order Perl
- And 55 other tutorials

LISA '06

20TH LARGE INSTALLATION
SYSTEM ADMINISTRATION CONFERENCE

A **Blueprint** for Real World
System Administration

DECEMBER 3–8, 2006 | WASHINGTON, D.C.

Keynote: Cory Doctorow, science fiction writer, co-editor of Boing Boing, and former Director of European Affairs for the EFF, on Hollywood's Secret War on Your NOC

Invited Talks by industry leaders discuss important and timely topics including:

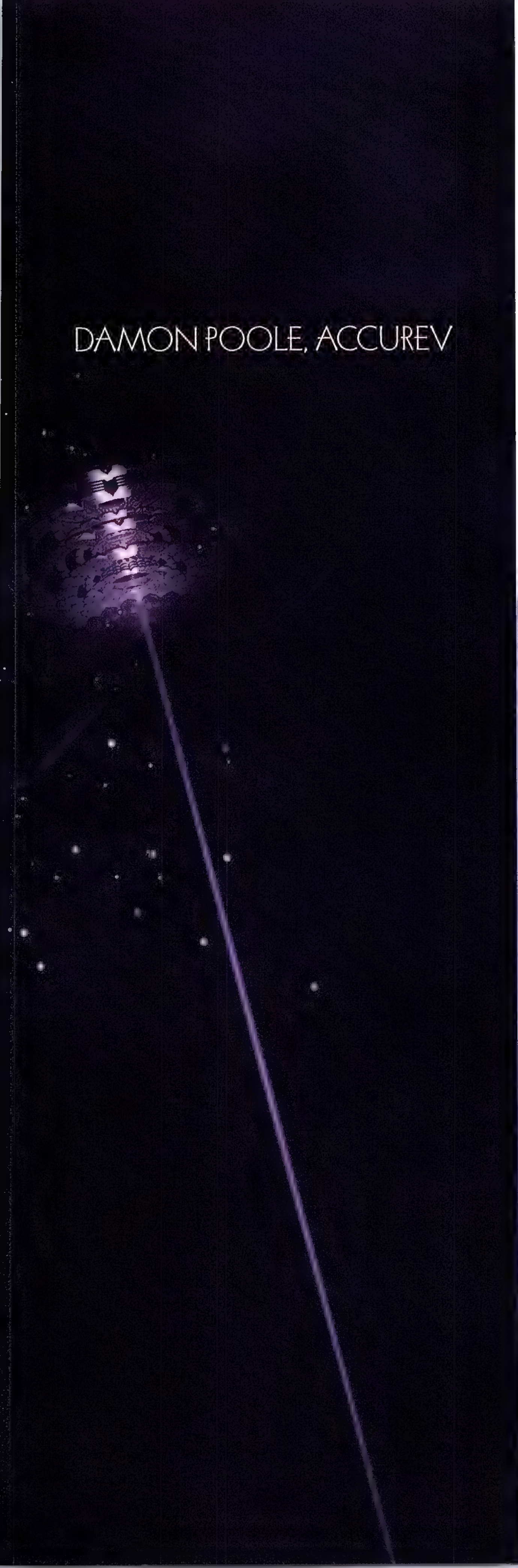
- Simple Nomad, Vernier Networks, Inc., "Corporate Security: A Hacker Perspective"
- DJ Byrne, Jet Propulsion Laboratory, "Open Source Software and Its Role in Space Exploration"
- Mazda Marvasti, Integrien, "Everything You Know About Monitoring Is Wrong"

Refereed Papers, Hit the Ground Running Track, Guru Is In Sessions, Vendor Exhibition, Workshops, BoFs, WiPs, and more!

Register by November 10 and save! www.usenix.org/lisa06/acq

LISA '06 offers the most in-depth, real-world system administration training available!

Breaking THE MAJOR RELEASE HABIT


DAMON POOLE, ACCUREV

Can agile development make your team more productive?

Keeping up with the rapid pace of change can be a daunting task. Just as you finally get your software working with a new technology to meet yesterday's requirements, a newer technology is introduced or a new business trend comes along to upset the apple cart. Whether your new challenge is Web services, SOA (service-oriented architecture), ESB (enterprise service bus), AJAX, Linux, the Sarbanes-Oxley Act, distributed development, outsourcing, or competitive pressure, there is an increasing need for development methodologies that help to shorten the development cycle time, respond to user needs faster, and increase quality all at the same time.

An emerging response to this challenge is a methodology called agile software development, the common theme of which is taking a traditional development process with a single deliverable at the end and splitting it into a series of small iterations, each of which is a microcosm of the full process and each producing working software.

Adopting an agile methodology poses its own set of challenges. It is used mostly by early adopters with small colocated teams, it has little tool support, and though the adoption can be done in phases, getting the full benefits of agile development requires sweeping changes to all phases of the project lifecycle. In addition, agile is actually a large and growing family of methodologies, including extreme programming, Scrum, lean software development, Crystal methodologies, feature-driven development, and many others. Although it is nice to have a wide selection, it does complicate the adoption process. Agile also challenges many fundamental and long-held legacy beliefs about the software development lifecycle. This makes it difficult to achieve the critical mass required to start an agile project.

Breaking THE MAJOR RELEASE HABIT

THE KEY TO AGILE: SHORT ITERATIONS

It has been widely stated that the benefits of agile development are producing value, maturity, and feedback faster; allowing early feedback to inform requirements to produce a better product; and increasing business flexibility. I believe that these benefits are all derived from a single agile development practice, short iterations, and that all other practices are enablers. Therefore, I will focus on short iterations.

The amount of time between releases varies widely from organization to organization and from project to project, but according to a Forrester Research report, "The Expanding Purview of Software Configuration Management" (July 2005), most iterations are still in the range of three to six months or more. No matter what your development process looks like, it is very likely that it can benefit from agile development. In many ways, what you are doing now undoubtedly overlaps with the agile practice of short iterations, whether you realize it or not. If you do one or two major releases per year, you probably don't think of yourself as doing short iterations. But hold on, what about all of those release candidates you produce near the end of a release cycle? Sure, you probably released only a couple of them externally, and even then they were betas, not "real" releases. And what about the three unplanned follow-on releases you did after that major release? How about hot fixes? Customer one-offs? Builds for QA? Demo builds?

If your customer is internal, you probably release once or twice per week or perhaps even per day. Before you start claiming to be agile, however, realize that frequent releases alone do not qualify as agile development.

You may think of all those unplanned and multiple candidate releases as exceptions or as impediments to producing high-quality software. Another way to look at it, however, is that change is an inevitable fact of life and it is better to embrace that change than try to reduce it. This is exactly what agile is all about, and it includes tools that help to transform what feels like hurtling at high speed out of control with danger at every turn into regularly scheduled coast-to-coast jet flights.

By using short iterations, you are able to get customer feedback sooner. Instead of releasing a major new feature all at one time, if you break it into logical pieces (assuming that this can be done for a particular feature), you can find out sooner which pieces are taking the feature in the right direction and which are not, and adjust your plans as you go.

Let's say that the perfect implementation of a new feature has 12 sub-components. Of course, you can't know in advance what that perfect implementation is. Using a traditional approach, you implement the first version in a year with 12 sub-components. It turns out that six are exactly what are needed, and six are completely wrong (even though that's what was asked for). So, a year later, you release the second version, and now you've got nine out of 12 correct...and so on.

Using short iterations, you could have released a component a month, getting feedback once a month instead of once a year. Using this tighter feedback loop, you should logically converge on the correct solution much faster than you would on a yearly schedule.

FREQUENT RELEASES AND QUALITY

The idea of frequent releases tends to be associated with low quality. It is easy to understand this line of reasoning, but it is a misunderstanding of agile development.

If you take a non-agile process and simply crank up the frequency of releases, you are guaranteed to take a hit in quality. Also, if your test phase is three months for a release, the idea of shrinking it down to a week so that you can release every month is pretty scary. But neither of these things is what actually happens. Nobody takes an existing process and simply starts releasing every month, and three months of testing isn't scaled back to just a week of testing.

If you have one release per year and go to a monthly release schedule, what really happens is that you redistribute the many tasks involved in a single release into 12 smaller releases. Each task—writing the spec, doing the design, writing the code, writing the automated tests, etc.—for a particular change still takes the same amount of time. The difference is that instead of occurring sometime during a one-year period, these tasks now occur over a one-month period. The exception is for changes that will take more than a month. In that case you have three choices: temporarily use a large enough iteration to accommodate the change, find a way to break the task up into smaller pieces that will fit into a one-month iteration, or do the change separately and add it to whichever iteration is in progress when the change occurs.

Some tasks, such as doing a full system build or packaging the software for release, are independent of the amount of change in a release. The biggest challenge is reducing the fixed overhead, which generally includes testing, so that the ratio of change to overhead stays at an acceptably high level.

CRITICAL PATH ANALYSIS

To do short iterations, everything needs to be adjusted to fit a shorter cycle time. An iteration is really just the time between releases. This is actually easier than it sounds. The key is to determine the critical path of a small development task (for example, a minor bug fix) from initiation to end product and then to determine the amount of time involved. You may be surprised by the result.

Let's consider a slightly different exercise. What is the critical path time for a customer-down hot fix in a high-impact area of the code? Your company probably has a special process for this situation, and once an issue has reached this status, it undoubtedly gets top priority and all of the resources it needs. Thus, there will be no dependencies on other issues, no waiting for somebody to get back from vacation, or any other delays in the critical path other than the time directly associated with the subtasks required to produce the hot fix. Since this involves a high-impact area of the code, it should get the highest level of scrutiny that your organization has to offer: multiple code reviews, regression tests, manual testing, the addition of new tests, stress testing, etc.

This type of hot fix typically takes four to eight hours with a worst case of 24 hours. Because time is critical for this issue, testing is limited to a subset of the testing that is done for a regular release. For example, only platform-specific testing or data integrity testing is done first; then the full battery of testing is done post-delivery, which might take another day. Most organizations can turn around a high-quality release that contains just a small change in two days or less if they really have to.

The critical path exercise just described demonstrates that the actual overhead associated with a single small change is at most two days. Since most of that overhead is related either to fully automated testing or to stepping through a manual test plan, and needs to be run through only once, why is it that so many organizations perceive that there is a much larger overhead—sometimes as much as three months for a one-year release cycle—associated with getting a product out the door?

It boils down to two simple answers: breaking the habit of long iterations is difficult, and long iterations hide fundamental problems. To make matters worse, these two problems tend to reinforce each other. The hidden problems help to keep the cycle time long, and the long cycle time helps to keep the problems hidden.

Since everybody "knows" that it takes a long time to qualify a major version of software for release, you of course want to get as much in there as possible. This desire must be balanced against the fact that if you take too long to get the next release out the door, you may fall behind your competitors or miss out on opportunities. As the planned release date comes and goes, you get nervous that it is taking too long and that you're going to break all of those promises you made, so you look for shortcuts. Eventually, the release goes out, but it has some problems and has to be patched a few times before settling down. Everybody remembers that and you vow that next time you won't take any shortcuts, thus reinforcing the belief that shortening the process produces lower quality.

LONG ITERATIONS HIDE PROBLEMS

On the one hand, producing a quality release takes two days of overhead. On the other hand, it takes three months. How can both of these statements be true? This is where the idea of "hidden problems" comes in. The many variations of hidden problems all stem from one root cause: Feedback on process problems comes late in the cycle—too late to take preventive action, so curative action has to be taken instead. Remember the old adage, "An ounce of prevention is worth a pound of cure."

Let's say that you have 100 planned work items for a release. Of course, the system test comes near the end of the cycle because you know it will be painful and you don't want to go through that process more than once. Therefore, you may do all of the specification and design up front for all of the changes, or you may do it in batches. Perhaps you then move on to doing all of the coding and testing, or maybe you start coding as the batches of specifications and designs are finished.

Regardless of the exact process, you won't start to get information about how good the requirements, specification, design, and coding are until the system test starts. Now you may find that the requirement gathering was done very poorly because the testers can't make heads or tails of how things are supposed to work. If, instead, you had broken the release into 10 iterations, you would have found this out after the first iteration, corrected it, and as a result the impact of the problem would have been reduced by a factor of 10. Instead, you now have to take more time than planned because of the rework required for many or most of the 100 work items, not just in the coding but also in revisiting the requirements, specifications, and design, and rewriting the test plans and automated tests.

The delays associated with process problems being detected late in the cycle occur every release, which is why so much time is reserved for the end game. It isn't the qualification of the release that takes so long—that still takes only two days. It is the process of getting the release to the point that it makes it through that two-day gauntlet, without any problems, that takes so long.

FEATURE CREEP

A year is a long time. During that time, the pressure to add more work items to the release grows and grows. After all, what are one or two more items among 100? Surely there will be time to fit them in during that year!

How soon we forget our arch-nemesis: feature creep. Before you know it, you've been seduced into adding an extra 30 work items to the plan and the code freeze date is upon you. Somehow you have ended up doing all of these small new items but you still haven't done some of the must-have items from the original plan. But that's OK; you still have those three months of testing, so you can get at least some of those items done then. You would like to have a high-quality release as close to on-time as possible. So, painful as it may be, you cut some of those must-have items from the plan and create a new follow-on release right after the main release.

Feature creep seems to be a function of the time between releases. The more features you add, the longer it takes to get the release done, the more opportunity you have for feature creep. A large set of features in a release, compounded by the additional changes required by discovering process problems late, and feature creep all work together to create two additional problems: code churn and increased difficulty of root-cause analysis.

Once system testing begins, there is enormous pressure to fix problems and meet the deadline. Many people are making many changes to interdependent systems. Just as module A, which depends on module B, starts to work with the previous set of changes made to module B, the next set of changes to B comes along and now A no longer works. Plus, the set of changes to A breaks C, and so on. Since so much is changing, it takes a long time to do root-cause analysis to figure out how to solve problems and make fixes. Sometimes it seems that for every problem solved, two new ones appear.

GETTING STARTED WITH AGILE DEVELOPMENT

Making changes and adopting new practices is hard. Anything that requires sweeping changes usually encounters a lot of resistance or is done only when the situation is so hopeless that changing it "couldn't possibly make things any worse."

I recommend a gradual adoption of agile development via a series of small process improvements. There is no need to start with a brand new project or to make wholesale changes in an existing project. It doesn't matter what process an organization is using, it can start down the path of agile development today and adopt as much of it as desired at whatever speed feels comfortable. Two enabling practices that make good starting points are writing tests first and work-item ranking (aka backlog).

Writing tests first. Writing a test requires having a solid understanding of how something is supposed to work so that you can verify that it is working properly and that it fails properly (among other things). Tests and requirements are closely related. If you are unable to write tests, that's a good early warning that you have a problem with the requirements, in which case, it is unlikely that anyone will be able to write the code that the tests are supposed to check. Since writing the tests should take less effort than writing the code, it makes sense to write the tests first as an early warning system for any requirements problems and to reduce the chance of having to rewrite code because of those problems.

Work-item ranking. Work-item ranking, also known as backlog in the Scrum agile methodology, is simply placing the outstanding work items in a list from highest priority to lowest priority from the perspective of your market. This practice can vastly simplify project planning, keep you focused on your target market, and help to prevent feature creep. If a new work item is suggested for the current iteration after work on the iteration has begun, then it is considered for the current iteration only if it ranks higher than a work item that is planned for the current iteration but has not yet been started.

AGILE'S POSITION IN THE TECHNOLOGY ADOPTION LIFECYCLE

Although agile development definitely has buzz and growing momentum, not that many people are actually

Agile Resources

To learn more about agile development, consult the following sources:

The Agile Alliance; http://www.agilealliance.org. This is a great place to start any search for more information about agile development.

Extreme Programming Explained, 2nd Edition, by Kent Beck (Addison-Wesley Professional, 2004). This is a seminal book on agile development. Whether you agree with the idea of 3x5 cards or not, this is a short and easy read that is well worth the effort.

The Toyota Way, by Jeffrey K. Liker (McGraw-Hill, 2003). Lean manufacturing, pioneered by the Toyota Production System, is one of the key influencers of agile thinking. In this fascinating book, Liker not only describes Toyota's approach to the lean methodology, but also explains how it is just one aspect of Toyota's total approach to producing high-quality products in an unusually short amount of time.

Lean Software Development, by Mary Poppendieck and Tom Poppendieck (McGraw-Hill, 2003). These folks have done an excellent job of applying the lessons of lean manufacturing to software development. Reading this book was the tipping point for me.

Agile Project Management with Scrum, by Ken Schwaber (Microsoft Press, 2004). As one of the very few "lessons-learned" books on agile, it relies heavily on case studies to illustrate the principles of Scrum as applied in a wide variety of scenarios. This is a good read regardless of which agile methodology appeals to you.

using it today. It still has the feel of being in the "early adopters" or even "innovators" phase of Geoffrey Moore's technology adoption lifecycle. Since there aren't many people using agile development, it will be difficult to find people in your existing network to lean on for help in adopting it. To get the benefits, you will have to sign up for the risk involved in being an early adopter and make a commitment to leading your first agile project to success.

The main reason the software industry exists in the first place is a fundamental belief that people will pay for the productivity gains created by automation. Yet, many proponents of agile development recommend keeping track of project tasks with 3x5 cards on a wall, with their position on the wall indicating their status. Unfortunately, this makes it difficult to take advantage of modern technology to edit, search, categorize, reorganize, report, track, manage, distribute, copy, share, access remotely, and do backups with the project's information. This in turn makes it difficult for team members to work in different offices, at home, or while traveling.

The reason for resorting to primitive tools is in part a reaction to the feeling that software process tools are bureaucratic in nature. There is some truth to that, as evidenced by the fact that most software tools were designed to fit into a framework of traditional development practices and have not yet adapted beyond superficial changes to meet the challenge of agile development.

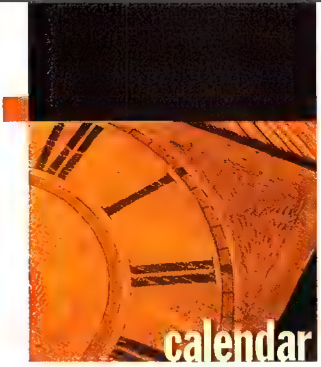
For example, to adopt the practice of work-item ranking, you will undoubtedly need to resort to using Excel, do some tooling of your own to support it, wait until the feature becomes available in your project management or issue tracking tool of choice, or bite the bullet and migrate to one of the small crop of new agile project management tools such as RallyDev or VersionOne.

SCALING AGILE BEYOND SMALL COLOCATED TEAMS

The biggest challenge slowing mainstream adoption of agile development is scaling it beyond small and colocated teams. In many of the agile methodologies, having a small colocated team is either a stated requirement or an underlying assumption. This combined with the deemphasis on automation and limited tooling means that larger and/or distributed teams will need either to experiment with adapting an agile methodology to work in their environment or to wait for others to finish blazing those trails. Q

LOVE IT, HATE IT? LET US KNOW
feedback@acmqueue.com or www.acmqueue.com/forums

DAMON POOLE is CTO and founder of AccuRev Inc. (http://www.accurev.com), an application lifecycle management (ALM) software company that develops software configuration management (SCM) tools focused on improving parallel and distributed development for global teams to remain agile and competitive. Poole has developed SCM best practices over his 15 years managing small and large development projects at companies such as Fidelity Investments and the Open Software Foundation, where he led the SCM tools development effort for the massive multivendor integration projects. Poole earned his B.S. in computer science at the University of Vermont in 1987.



OCTOBER

Grace Hopper Celebration of Women in Computing Conference
Oct. 4-7, 2006
San Diego, California
http://www.gracehopper.org/

Software Architecture Design and Analysis
Oct. 10-11, 2006
Pittsburgh, Pennsylvania
http://www.sei.cmu.edu/products/courses/saad.html

Strategy to Reality: Securing the Converged Enterprise
Oct. 10-11, 2006
Washington, D.C.
http://www.ip3seminars.com/s2r.htm

VERIFY 2006 International Software Test Conference
Oct. 10-11, 2006
Washington, D.C.
http://www.effectivesoftwaretesting.com/Conference_Verify.aspx

STARWEST (Software Testing Analysis and Review)
Oct. 16-20, 2006
Anaheim, California
http://www.sqe.com/starwest/

The DC PHP Conference
Oct. 19-20, 2006
Washington, D.C.
http://www.dcphpconference.com/

Oracle Open World
Oct. 22-26, 2006
San Francisco, California
http://www.oracle.com/openworld/index.html

VSLive!
Oct. 24-27, 2006
Boston, Massachusetts
http://www.ftponline.com/conferences/vslive/2006/boston/

Serious Games Summit
Oct. 30-31, 2006
Arlington, Virginia
http://www.seriousgamessummit.com/

Zend/PHP Conference and Expo
Oct. 30-Nov. 2, 2006
San Jose, California
http://zendcon06.kbconferences.com/

CCS (ACM Conference on Computer and Communications Security)
Oct. 30-Nov. 3, 2006
Alexandria, Virginia
http://www.acm.org/sigs/sigsac/ccs/CCS2006/

NOVEMBER

CIKM (Conference on Information and Knowledge Management)
Nov. 5-11, 2006
Arlington, Virginia
http://campus.acm.org/calendar/confpage.cfm?ConfID=2006-7772

OSDI (Usenix Symposium on Operating Systems Design and Implementation)
Nov. 6-8, 2006
Seattle, Washington
http://www.usenix.org/events/osdi06/

IEEE International Symposium on Software Reliability Engineering
Nov. 7-10, 2006
Raleigh, North Carolina
http://www.csc2.ncsu.edu/conferences/issre/

To announce an event, E-MAIL QUEUE-ED@ACM.ORG OR FAX +1-212-944-1318

Web 2.0 Conference
Nov. 7-9, 2006
San Francisco, California
http://web2con.com/

Six Sigma in Software and IT Conference
Nov. 7-9, 2006
San Francisco, California
http://www.wcbf.com/quality/5068/

Internationalization and Unicode Conference 30
Nov. 15-17, 2006
Washington, D.C.
http://www.unicodeconference.org/

DECEMBER

First International Workshop on Mobility in the Evolving Internet Architecture
Dec. 1, 2006
San Francisco, California
http://user.informatik.uni-goettingen.de/~mobiarch/

LISA (Large Installation System Administration Conference)
Dec. 3-8, 2006
Washington, D.C.
http://www.usenix.org/events/lisa06/

ICSOC (International Conference on Service Oriented Computing)
Dec. 4-7, 2006
Chicago, Illinois
http://www.icsoc.org/

CLASSIFIED

Epic Systems Corporation

Problem Solver/ Engineer/Software Analyst

Combine your problem solving skills and technical interest with our training to support our healthcare clients. Participate in analysis, training, quality assurance, troubleshooting, implementation and support. Apply at www.epicsystems.com.

International Christian University

Faculty Position in Computer Science

The International Christian University in Tokyo invites applications for a full-time tenure-track faculty position in Computer Science, starting date April 1st, 2007. Please see http://science.icu.ac.jp/csc/ for all details

Korea University

Contract professor/ Post Doctorial Researcher

The department of Electrical Engineering in Korea University is seeking for contract professor or post-doc. In all fields of electrical, electronics, and electro wave research areas. Visit http://ee.korea.ac.kr/job.htm for details.

Ludic Labs, Inc.

Senior Software Multi-media R&D—Startup

Creative software R&D opening at pioneering startup. Must be proficient in C++, Java, internet, multimedia, with track record of success at interesting projects. As member of founding R&D team, will participate in all aspects of company—research, design, development & strategy. Mobile experience desired. B.S., M.S., Ph.D. in CS or related field.

St. Lawrence University

Assistant Professor—Tenure Track

St. Lawrence University invites applications for a tenure-track position in computer science in the Department of Mathematics, Computer Science and Statistics at the assistant professor level to begin in August 2007. Qualifications include a PhD in computer science or a related field, evidence of research potential, a strong commitment to teaching undergraduates in a liberal arts setting, and evidence of excellence in teaching. The teaching load is three courses per semester. Please visit the department's web page at http://it.stlawu.edu/~math for more information about our students and programs. Review of applications will begin on December 1, 2006, and will continue until the position is filled. Candidates should send a letter of application, a CV, a statement of teaching philosophy, and arrange for three letters of recommendation to be sent to: Ed Harcourt, Computer Science Search Committee Chair, Department of Mathematics, Computer Science and Statistics, St. Lawrence University, Canton, NY 13617. St. Lawrence University, (http://www.stlawu.edu) chartered in 1856, is an independent, private, non-denominational university located in Northern New York between the St. Lawrence River and the Adirondack Park. The Canton-Potsdam area is home to three other universities; Clarkson University, SUNY Potsdam, and SUNY Canton. For more information visit, http://www.stlawu.edu/resources/job.html. St. Lawrence University is an Affirmative Action, Equal Employment Opportunity employer. Women, minorities, veterans, and persons with disabilities are encouraged to apply.

Continued from page 56

painstakingly reading the very rare sources not yet online. Perhaps the record is the pushing back overnight, in one fell swoop, of 76 *OED* botanical citations when in 1976 French lexicographer Claude Boisson discovered a 17th-century Spanish herbal. I'll drink to that![4]

So, yes, there are professional citationologists who earn their daily-donnish, inchworm bread seeking authorial precedences. When asked by their kids, "Mummy/Daddy, what do you do for a living?" they might reply, "I believe that question was first posed by F. Josephus Jr., circa 67 CE, although there are good scholars who place it much earlier in the Shang dynasty...why aren't you in bed?" The JobSpec template is, informally: Who *first* said <T>, how, when, where, and (holiday bonus), why?

Finding the first usage of *citationology* requires, somewhat recursively, its own methods. Appending the Grecian formulaic *-ology* to indicate the scientific study of the suffixed stem is not in itself a shattering neological achievement. But establishing an accurate space-time-stamp for the emergence of a particular *-ology* can be useful to science historians. I seem to have been the first to use *citationeering* (*Unix Review*, March 1989—but submitted January 1989!) as a rather derogatory dig at dilettante citation hunters (I have since dubbed them, unfairly, Wikipedophiles). The first appearance of *citationology* that I can find is an authoritative claim by Eugene Garfield (publisher *of The Scientist* magazine): "I offer the term *citationology* as the theory and practice of citation, including its derivative disciplines, citation analysis and bibliometrics," submitted April 9, 1998.[5] It's rather exciting to find the expert practitioner naming an extension to his own domain, and with strong evidence ("I offer the term...") of his priority. Nevertheless, priority always remains an open case.

Search engines such as Web of Science, Google Scholar, and Scopus permit endless refinements in the citation game. Careers, tenures, and funding can depend on not only establishing publishing priorities but also comparing "citedness scores" (who has been quoting or referencing your papers) and deducing an author's or paper's "impact." Note, of course, that "I love Chomsky" and "I hate Chomsky" both register a match for "Chomsky".[6]

What distinguishes the citationologist from the citationeer is not only a mature, finely honed skepticism about the commercial search-engine distortions mentioned earlier, but also a balanced cynicism about the uneven quality of Web *content*. We must be equally suspicious of all our sources, of course, whether in bound-paper books and journals, or on disks and screens. But it is often more difficult to authenticate the volatile media even though most of our information actually originates and is exchanged electronically. Our Web sites and mailboxes include peer-reviewed repositories of mankind's accumulated knowledge intermingled with suppositories of willful or accidental disinformation. As one tech writer's prologue warned: "Believe only the true sections. Ignore the rest."

AUTHOR! AUTHOR!

Let's not be too grumpy. There's a growing citation and pop-grammarian industry to amuse the lay wordsmithy. I've counted six books with *Lost for Words* in their titles. Anthologies of "familiar quotations," pioneered by John Bartlett, abide in print and online. Our very own ACM Web site maintains a list of computer-related sayings and witticisms. Kevin G. Barkes keeps an unusually varied collection at http://www.goodquotations.com. In the BBC Radio quiz "Quote/Unquote," the strategy of contestants who cannot immediately identify a given quotation seems to be a plausible guess based on genre. The Bible, Shakespeare, or Churchill are statistically promising for somber and majestic pronouncements, while for modern

As one tech writer's prologue warned, **"Believe only the true sections,** ignore the rest."

wisecracks, try Oscar Wilde, Mark Twain, or Will Rogers, with a side bet on Dorothy Parker. The wildcard, catchall "no match" is the ageless polymath Anon, accidentally tenured as Professor Anon. Thereby hangs a warning tale of search-engine hazards.

The "Professor Anon," whom I thought I had invented as the obvious putative source of all otherwise unascribed wisdom, turns up more than 70 matches when Googled. Ignoring the blogger who signs off as Anonymous and is then addressed sarcastically as "Professor Anonymous," and overlooking a Dr A. N. Other and Prof Anon at the (possibly?) mock University of Anon, Scotland, we do finally find a real live don. He is referenced as Professor Anon Monshouwer, a leading educologist (you can look that up—it's not quite an educationalist) at the University of Nijmegan, The Netherlands. Funny, methinks. Further research solves the mystery: He is, in fact, Anton Monshouwer. A spelling error in a Web paper on the history of educology spawns other references to my supposed phantom expert. You will all no doubt have your own pet encounters with the Web's high noise-signal ratio.

Ideally, we should examine all sources: glyph, manuscript, print, and audiovisual in every available language (not to mention the Latin motto tattooed on Angelina Jolie's torso).[7] Matching sounds and pictures is, we are promised, "on the way!" We must also allow for semantic variations; seek the gist rather than matching exact phrases. Aye, there's a major rub.

My lead-in quotation, for example, was assigned by Charlie Zimmerman to comedian Jack Benny, who actually quipped, "If I can't take it with me, I refuse to go," which is a close enough match for humans even if it presents a challenge to automated search engines. But, pause, which, if any, of Jack Benny's many scriptwriters might have prior claim? Likewise, who penned Fred Allen's less-famous response, "If you could take it with you, it would melt!" Naive grepping for literal string matches, however, would not fully resolve or explicate the Jack Benny quote. What is also needed is the wider context, starting, say, with the earlier post-Depression catchphrase, "You can't take it with you," popularized by George S. Kaufman's 1937 novel and Frank Capra's movie version. This is complex context, spelling hedonism to some: Live it up while you can, with a hint that you may not be "going" anywhere. Compare the cynical Union hymn, "Work and pray, live on hay; you'll get pie in the sky when you die."

Others, though, can read it as a warning against consumerism and as supporting Bill Gates's welcomed philanthropy, which derives from the aforementioned consumerism. I can live with this paradox, having visited the wonderful new (2001) William Gates Computer Laboratory, Cambridge, that replaces my 1950s EDSAC math lab haunts on the old Cavendish site. The Christian message is that such good deeds are rewarded: "You can't take it with you, but you can mail it ahead."

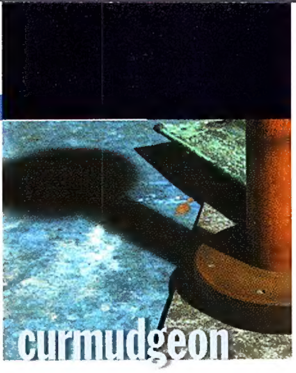
One can also find personal-computing interpretations of the Jack Benny gag. There are echoes of Adam Osbourne's first "portable" PC: "You *can* take it with you!" (well, with the help of Hernia, the Goddess of Weightlifters). With the advent of the mobile-phone-PDA-MP3-camera-Web-terminal, we move on to the portability of the credit card: "Don't leave home without it." We know who holds the copyright, but who said it first? Q

REFERENCES

1. When I first used this pun (*Unix Review*, September 1985), it was edited as "easier said than done." My jokey "IT laxicon" also suffers textual harassment, restored to "lexicon" as recently as my July/August 2006 Curmudgeon column in *ACM Queue*!
2. Like Yogi's baseball, with its curves, knuckle, and spitballs, cricket has a complex taxonomy of wicked deliveries (off-spin, leg-spin, yorkers, doozies) that has fascinated even pure mathematicians such as G. H. Hardy.
3. Winchester, S. 1998. *The Surgeon of Crowthorne*, London, Viking. The U.S. version was more enticingly titled *The Professor and the Madman*, HarperCollins, 1998. The *OED* still enlists help from outside readers. Details of the North American Reading Program: http://www.rice.edu/oed/readers.html.
4. Boisson, C. Earlier quotations for Amerindian loanwords in English. *OUP: International Journal of Lexicography* 1(4). The star in Boisson's list is *totora*, a South American plant that will be more familiar to you as the *Typha domingensis*. The *OED*'s 1936 citation for *totora* gets rolled back all the way to 1604!
5. Garfield, E. 1998. Random thoughts on citationology, its theory and practice. *Scientometrics* 43(1).
6. The infamous "selected quote" in show biz is relevant here, and worth a mention. The billboard says, "Incredible!—NY Times"; the critic has written, "I find it incredible that any sane person would pay money to see this show." And "Don't Miss It!" was extracted from "If the show's running late and you have a train to catch, don't miss it."
7. Subject to many palimpsestuous modifications: *Quod me nutrit me destruit* (What nourishes me also destroys).

LOVE IT, HATE IT? LET US KNOW
feedback@acmqueue.com or www.acmqueue.com/forums

STAN KELLY-BOOTLE (http://www.feniks.com/skb/; http://www.sarcheck.com), born in Liverpool, England, read pure mathematics at Cambridge in the 1950s before tackling the impurities of computer science on the pioneering EDSAC I. His many books include *The Devil's DP Dictionary* (McGraw-Hill, 1981), *Understanding Unix* (Sybex, 1994), and the recent e-book *Computer Language—The Stan Kelly-Bootle Reader* (http://tinyurl.com/ab68). *Software Development Magazine* has named him as the first recipient of the new annual Stan Kelly-Bootle ElecTech Award for his "lifetime achievements in technology and letters." Neither Nobel nor Turing achieved such prized eponymous recognition. Under his nom-de-folk, Stan Kelly, he has enjoyed a parallel career as a singer and songwriter.



You Can Look It Up—Or Maybe Not

Stan Kelly-Bootle, Author

Many are said to have said, "If I can't take it with me, I'm not going!" I've just said it, but that hardly counts. Who, we demand, said or wrote it first? It's what I call (and claim first rights on) a FUQ (frequently unanswerable question, pronounced *fook* to avoid ambiguity and altercation). Yogi Berra's famous advice was "You can look it up," meaning, in fact, "Take my word on this." He knew quite well that few had the means or patience to wade through the records. Nowadays, of course, as we quip in Unix, it's easier done than sed.[1] The portmanteau *wep* for grepping the Web, now realized and refined in countless search engines, lets us take up Yogi's challenge at face value with a few simplistic keystrokes and mouse-clicks. Yet, as I aim to indicate, life—at least the life of serious scholarship—is not a bowl of Googles.

Seeking earliest attributions has serious applications for etymologists and sociolinguists. Related questions of establishing priorities also loom large, litigable, and expensive in patents and intellectual property disputes. Indeed, it has emerged as part of a newish science called citationology, of which more anon. Take the proper noun *Google* and its derived parts of speech: verbal ("As I was a-Googling"); adjectival ("Google morality"); improper noun ("There's nothing like a nice Google."). You may think, with good reason, that the origins of *Google* have something to do with the incredibly large number googol (the 10^{100} spurious matches planted by advertisers), or with the act of goggling (the bulging or squinting [Middle English] eyes as the spurious matches scroll by). My own preferred etymology is the googly, a sneakily-bowled cricket ball that spins in the opposite direction to that expected by the batter.[2] All of which hints at the perceived dangers of naively scanning the Web. Even with enlightened Boolean modifiers, literal `strcmp()`-type char-string matches are, indeed, too damned literal. As Dr. Walter Martin (the original Bible Answer Man) used to say, "Text without context is pretext." And context cannot yet be conveniently, decently parsed and automated.

Further, the commercial search engines (show no surprise) have commercial agendas, meta-matches, and sponsored, hyped hyperlinks. These ploys can be overt and reasonable (as someone said, "No free lunch!"); others are hidden and insidious. "Galileo" may take you to Amazon's books on the guy (sort of acceptable) or to some dubious agent offering cheap flights to Pisa (sort of unacceptable, especially if it's the first match displayed). Browsing, formerly the idle nibbling of grass or the leisurely reading of books, is now the frantic chaining through endless, mislabeled nodes.

Chasing citations

THROUGH ENDLESS, MISLABELED NODES

Citations have been tracked and dissected for centuries by dusty, manual dictionary-makers—recall Samuel Johnson's self-definition of lexicographer as a "harmless drudge"? The classic case is the agonizing (1879-1928) compilation of the first *OED* (*Oxford English Dictionary*), started by Sir James Murray (1837-1915), who strove to supply the best-available and first-known citations for each headword as an essential supplement to the proposed etymologies and shifting semantics. Murray's way-out outsourced army of readers, flooding the *OED* HQ with millions of hand-written 4 x 6 "dictionary cards," included some truly paranoid word-tracers. There was the convicted "Lambeth murderer," American-born Dr. William Chester Minor, who supplied 20 years of wondrous citations from his library-cell in the Broadmoor Asylum for the Criminally Insane. And you must love Herbert Coleridge, another of Murray's fanatical helpers, "whose last words, at the age of thirty-one, were 'I must begin Sanskrit tomorrow.'"[3]

The citation chase involves years of bleary-eyed rummaging through rare manuscripts, books of all kinds—banned and available—newspapers, glyphs, tombstones, ostracons, and buried genizoth (Hebrew plural of *genizah*, a repository of discarded, damaged holy books). The introduction of computers and the gradual digitization of many of these sources clearly increased accuracy and reduced the drudgery, yet the nirvana of "beating" the *OED* by predating any of its "first" citations is still more likely to be attained by old-fashioned grunt work than by electronic searches. Anyone with minimal skills and a modem can scan the available databases. But the news-making predatings are those made from locating and

Continued on page 54

"A conference with something for everyone."
— Scott L. Boudreau, QA Manager, Tribune Interactive

Announcing The Third Annual

Software Test & Performance CONFERENCE

- **OPTIMIZE** Your Web Testing Strategies
- **LEARN** How to Apply Proven Software Test Methodologies
- WORK With Other Test/QA & Development essionals
- etter Investment From
- From Your

November 7-9, 2006

The Hyatt Reg

Cambridge,

Register by October 20 to take advantage of the Early Bird Rate!

OUNCE LABS
PARASOFT
SQA

Silver Sponsor

COMPUWARE.
SHUNRA

Media Sponsors

Software Test & Performance
SD Times
queue
ECLIPSE review
Software
BETTER SOFTWARE MAGAZINE
TEST QA Report
Methods & Tools

"If you get one impact idea from a conference it pays for itself. I got several at the ST&P Conference."
— Patrick Higgins, Senior Software Test Engineer, SSAI

Innovations by **InterSystems**

Java Developers Have Caché

The Object Database With Jalapeño. Persist POJOs With Ño Mapping.

InterSystems CACHÉ®

The object database that runs SQL faster than relational databases now comes with InterSystems Jalapeño™ technology that eliminates mapping. Download a free, fully functional, non-expiring copy at:
www.InterSystems.com/Jalapeno3S